PORTABLE DHCP SERVER
TECHNICAL REFERENCE

interniche
technologies, inc.

51 E Campbell Ave
Suite 160
Campbell, CA. 95008

email: support@iniche.com
http://www.iniche.com
support: 408.540.1160
fax: 408.540.1161

Rev-10.2005

Portions of the InterNiche source code are provided under the copyright of the respective owners and are also acknowledged in the appropriate source files:

Trademarks

TABLE OF CONTENTS

1. OVERVIEW

This Technical reference is provided with the InterNiche Portable Dynamic Host Configuration Protocol (DHCP) server. The purpose of this Document is to provide enough information so that a moderately experienced “C” programmer with a reasonable understanding of TCP/IP protocols can port the InterNiche Server to a new environment.

It is assumed that the one of the InterNiche DHCP server “Demo” sources are available as a reference. Demos are currently available for MS-DOS systems, running on top of InterNiche’s lightweight UDP API; or on Windows 95, where the demo DHCP server runs as a Windows application and uses the native Windows WinSock TCP/IP. The Demo executable requires a PC running MS-DOS 3.0 or newer and access to Ethernet via an ODI driver or an FTP Software type "Packet Driver". On request demos can be provided which implement token ring or PPP over a dialup link.

1.1 Terms and Conventions

In this document, the term “stack”, when used without other qualification, means the InterNiche TCP/IP and related code as ported to an embedded system. “System” refers to your embedded system. “Sockets” refers to the TCP API developed for UNIX at U.C. Berkeley. A “user” or “porting engineer” usually refers to the engineer who is porting the server software. An “end user” refers to the person who ultimately ends up using the “user’s” product. “FCS” is an acronym for “First Customer Ship”, the point in the software development cycle when the product is declared ready to ship. A “packet” is a sequence of bytes sent on network hardware, also known as a “frame” or a “datagram”.

Names of files, C structures and C routines are displayed as follows: **c_routine()**.

Samples of source code from C programs are displayed in these boxes:

```
/* C source file - the world’s 1 millionth hello program. */
main()
{
    printf(“hello world.\n”);
}
```

1.1.1 A Note About DOS

We apologize in advance to those engineers who resent our distribution and documentation seeming PC-DOS or Intel x86 oriented. The DHCP server has been ported to numerous CPUs and operating systems and has no internal DOS or Intel dependencies.

The DOS orientation is not because out engineers are enamored of DOS or the x86’s technical elegance (they are not), but simply a matter of market realities. There are more embedded systems being developed on and for Intel x86 chips than any other, and more development environments based on DOS and Windows than any other platform. Every client we’ve had has had at least one PC to unzip the files, read the documentation, and build the demo package. And PC based C Compilers and source level debuggers are affordable and varied. An additional argument for using DOS to develop an embedded systems product is its lack of advanced features. We've yet to encounter an RTOS vendor who asserts their product has weaker multitasking or memory management features than DOS. By demonstrating our products turning in excellent performance on plain vanilla DOS, we show that the stack has only minimal requirements from its host system.

1.2 What Does DHCP Do?

DHCP stands for Dynamic Host Configuration Protocol. It is designed to ease configuration management of large networks by allowing the network administrator to collect all the IP hosts "soft" configuration information into a single computer. This includes IP address, name, gateway, and default servers. There are about 50 if these information items which can be assigned with DHCP, and DHCP is designed so that "custom" configuration items can be added easily.

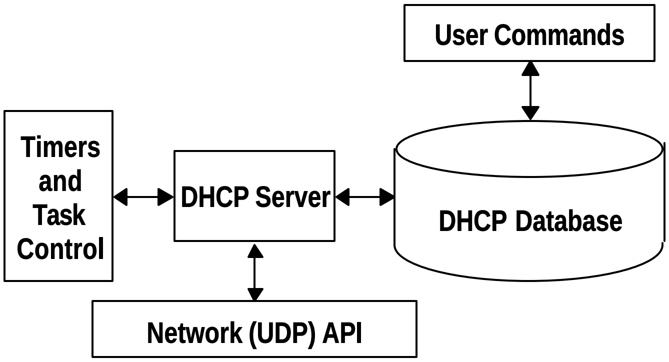
DHCP is a "client/server" protocol, meaning that machine with the DHCP database "serves" requests from DHCP clients. The clients typically initiate the transaction by requesting an IP address and perhaps other information from the server. The server looks up the client in its database, usually by the client's media address, and assigns the requested fields. Clients do not always need to be in the server's database. If an unknown client submits a request, the server may optionally assign the client a free IP address from a "pool" of free addresses kept for this purpose. The server may also assign the client default information of the local network, such as the default gateway, the DNS server, and routing information.

When the IP addresses is assigned, it is "leased" to the client for a finite amount of time. The DHCP client needs to keep track of this lease time, and obtain a lease extension from the server before the lease time runs out. Once the lease has elapsed, the client should not send any more IP packets (except DHCP requests) until he get another address. This approach allows computers (such as laptops or factory floor monitors) which will not be permanently attached to the network to share IP addresses and not hog them when they are not using the net.

1.3 And a Bit About BOOTP

In both this manual and other DHCP literature, you will find numerous cross references to BOOTP. BOOTP is/was the protocol which preceded DHCP in the TCP/IP world, and in fact DHCP is just s superset of BOOTP. The main differences between the two are the lease concept, which was created for DHCP, and the ability to assigned addresses from a pool. The InterNiche DHCP client (sold as part of our TCP/IP stack) can work with old fashioned BOOTP servers, and the InterNiche DHCP server can serve IP addresses to old BOOTP clients.

1.4 What Is a Port?



While processing DHCP requests, the server will consult a local database. This database is generally set up by an end user and contains the IP address pools, default information for clients, and perhaps special configurations for specific clients. This database may be stored in a traditional disk file or in flash. On some embedded systems it is even practical to ship the server with a factory configured database, such as systems containing InterNiche's NAT Router code.

In the world of portable stacks, the stack designer does not know what tasking system, user applications, or interfaces will be supported in the target system. So a "portable" stack is one that is designed with simple, generic interfaces in these areas, and a "glue" layer is created which maps this generic interface into the specific interfaces available on the target system. Again using the example of sending a packet, the stack would be designed with a generic **send_packet()** call, and the porting engineer would code a "glue" routine to send the packet on the target system's network interface.

Making a stack portable involves minimizing the number of calls which have to go across glue routines, and keeping the glue routines simple and therefore easy to implement. The glue routines also need to be well documented. The interfaces to the InterNiche DHCP have evolved through years of porting to a variety of processors, network media, and tasking systems. Wherever possible we have used standard interfaces (e.g. Sockets, ANSI C library) or included glue routines to illustrate their use.

The bulk of the work in porting a stack is understanding and implementing these glue routines. The InterNiche DHCP server has two kinds of glue routines: database access and network access.

The calls to set and extract database items are "abstracted out", which means they are generic calls which will need to be provided as part of the port. Files are provided which use standard file IO calls (**fopen**, **fread**, etc) to implement a fully functional database, so if your target system has a disk-like device you will most likely be able to use this code as is. If your database will be kept in flash memory (and it does not have a file-system like API), you will need to develop you own data structures and the code to access them.

The other set of glue routines needed for a port is the network access. If you are using the DHCP server with the InterNiche IP stack, you are in luck: a file is provided which implement the required routines on the lightweight API. We also provide a **Sockets** based glue layer for DHCP server. Most of our customers can use one of these two layers. The rest will need to provide some simple routines to allow the server access to their UDP protocol.

1.5 Requirements

Before beginning a port, the programmer should ensure that the necessary resources are available in the target environment. Here is a brief summary of services InterNiche DHCP server needs from the system:

- Access to a UDP layer, with "listen" capabilities (e.g. Sockets)
- A timer which ticks at least once a second.
- A non-volatile read/write method for storing database items (e.g. disk or flash memory)
- Memory as described below.

1.5.1 Memory Requirements

There is no easy way to determine the exact memory sizes required, however a rough idea can be obtained by examining the DOS DEMO executables. Some figures are given below. This program is compiled for the Intel 8088 processor with Microsoft C 8.0, using default optimization options. It implements the DHCP server, NAT routing IP, TCP, and a Web Server on a single ODI Ethernet driver. For this configuration, DHCP server memory sizes are as follows (these figures are subject to change without notice, but are current as of 1/21/98):

BYTES	DESCRIPTION
8489	DHCP "core" server code
3487	DHCP file IO code (**dhcpsnv.c**)
332	Menu front end routines (w/o generic menu drivers)
693	**dhcpport.c** file (UDP IO glue layer)
13001	**Total size DHCP Server code added to IP Stack**

These sizes may also improve slightly when compiled for an Intel 186 or 386 chip. They generally worsen when ported to a RISC processor. Since TCP uses the largest portion of memory, systems needing only IP and UDP services (e.g. a dedicated DHCP server) can remove the TCP layer and save considerable memory.

1.5.2 Operating System Requirements

The DHCP server also requires a few basic services from the operating system. These are listed here:

clock tick	- The DHCP server needs to be called once a second to free resources for addresses which have timed out.
memory access	- The standard **calloc()** and **free()** library calls are ideal, however DHCP server can also be mapped to a "partition" based system with very little effort.

1.6 Demo Package Directories List

The InterNiche sources are typically distributed as a DOS zip file, **dhcpsrc.zip**. This file should be unzipped with **pkunzip** (or compatible utility) in such a way as to preserve the underlying directory structure. It includes the required libraries, makefiles, and include files to implement a full featured DHCP server on a DOS PC which has an Ethernet adapter. Build tools can be either Microsoft C for DOS version 8, or Borland C for DOS version 4.5.

On DOS, the command to unpack the code is:

```
c:> pkunzip -d dhcpsrc.zip
```

TAR and **gzip** versions are also available.

Assuming you've unpacked in a directory named **\demo**, the file unzip should create the following directories within the **\demo** directory:

dosmain	- **main()** routine, console and menu handling, and other files specific to the demo application.
dhcpsrv	- The DHCP server sources and include files listed below.
inet	- IP, UDP, and related sources (excluding TCP). Includes startup, interface, and buffer management code.
misclib	- demo menu system, utility routines, and some file IO.

Other sub-directories may be present depending on your target system, however these contain no code required for the DHCP server per-se.

Please refer to **toolnote.doc** for information about building the sources.

2. STEP BY STEP PORTING GUIDE

The section describes the steps needed to port the InterNiche DHCP server to a new environment. The discussions below generally assume that the stack is being ported to a small or embedded system with a Sockets API interface and that a minimal ANSI C library is available.

The recommended steps to getting the server working on your target system are as follows:

1. Copy the portable source files into your development environment.
2. Create your version of **dhcpport.h** and compile portable sources.
3. Code your glue layers (**dhcpport.c**) and compile.
4. Build a system, test, and debug.

2.1 Port Dependent Files

Before beginning step one, you should be aware of which files in the InterNiche distribution are the "portable" files and which are not. The portable files are those which should be compiled and used on any target system without modification. The unportable, or "port dependent" files, are those which will need to be replaced or heavily modified for different target systems. The following is a list of DHCP server source files which should NOT need to be modified in the course of a normal port. If you feel you need to modify one of these files in the course of a routine port, please discuss it with InterNiche's technical support staff first, so we can either suggest an alternative, or modify our sources to reflect the change.

2.2 Source Files List

The portable DHCP server source files. **Do Not modify.**

dhcpsrv.c - the kernel of the DHCP server
dhcpsrv.h
dhcp.h

The database IO files: may need modification on some ports:

dhcpsnv.c - file system database IO

The network (Sockets) glue files:

dhcpport.c - sockets and other system dependent calls

InterNiche menu system routines - can be used as is with InterNiche menuing system, else may be replaced or omitted:

dhcpmenu.c - routines to dump statistics and status

The per port files - Usually need to be re-written for target system:

dhcpport.h - memory allocation, time ticks, etc.

2.3 The Master DHCP Port File: dhcpport.h

Before you compile these files you should create a version of the file **dhcpport.h**. This file contains most of the port dependent definitions in the stack. CPU architectures (big vs. little endian), compiler idiosyncrasies, and optional features (such as multiple interfaces or a 10-net only configuration) are controlled in this file. A single mistake in this file (such as getting big and little endian confused) will guarantee that your port won't work properly. Taking a few hours up front to implement the file line by line is time well spent. This section outlines the basic contents of **dhcpport.h**.

2.3.1 Standard Macros and Definitions

The InterNiche DHCP expects **TRUE**, **FALSE**, and **NULL** to be defined within the scope of **dhcpport.h**. The best way to do this is usually to include the standard C library file **stdio.h** inside **dhcpport.h**. If **stdio.h** is impractical to use or missing, the examples below will work for almost every C environment:

```
#ifndef TRUE
#define TRUE -1
#define FALSE 0
#endif
#ifndef NULL
#define NULL (void*)0;
#endif
```

2.3.2 Memory Allocation

The DHCP server code allocates and frees memory blocks dynamically as it runs. It uses the macros listed below to do this. If your target system supports standard C **calloc()** and **free()**, the macros map directly as follows:

```
/* map malloc and free to system calls */
#define DHCP_ALLOC(size)       calloc(1, size) /* DHCP header alloc/free */
#define DHCP_FREE(ptr)         free(ptr)
#define DHPOOL_ALLOC(size)     calloc(1, size) /* DHCP free pool list alloc/free */
#define DHPOOL_FREE(ptr)       free(ptr)
#define DHENT_ALLOC(size)      calloc(1, size)          /* DHCP entry list
alloc/free */
#define DHENT_FREE(ptr)        free(ptr)
```

Many RTOS systems do not use calloc due to performance issues. Generally, they use a system which supports allocations of fixed size "partitions" (blocks) instead. The macros above are designed to support this - each of the **_ALLOC** macros only allocates a single size. Thus the macros can be mapped to a call to allocate the next largest partition size.

2.3.3 CPU Architecture

Four common macros are used from Berkeley UNIX for doing byte order conversions between different CPU architecture types. These are **htons()**, **htonl()**, **ntohs()**, and **ntohl()**.

They may be either macros or functions. They accept 16 and 32 bit quantities as shown, and convert them from network format ("big-endian") to the local CPU's format.

Most IP stacks already have these byte ordering macros defined. If this is the case you should try to find the existing include file which defines them and use it rather than duplicate them. The information below is for the rare situations where these macros are not already available.

For Motorola 68000 family and most RISC chips, these can just return the variable passed, as in this example:

```
#define htonl(long_var)     (long_var)
#define htons(short_var)  (short_var)
#define ntohl(long_var)     (long_var)
#define ntohs(short_var)  (short_var)
```

The Intel 8086 and its descendants require the byte order in the word or long to be swapped. The **lswap()** and **bswap()** primitives provided with the InterNiche DEMO package can be used as illustrated here:

```
#define htonl(long_var)     lswap(long_var)
#define htons(short_var)  bswap(short_var)
#define ntohl(long_var)     lswap(long_var)
#define ntohs(short_var)  bswap(short_var)
```

2.3.4 Debugging Aids

dtrap() is a macro called by the DHCP code whenever it detects a situation which should not be occurring. The intention is for the **dtrap()** routine or macro to try to trap to whatever debugger may be in use by the programmer. Think of it as an embedded break point. For most Intel x86 processor debuggers, this can be done with an **int 3** opcode. The macro below is effective if your Intel C compiler accepts inline assembly:

```
#define dtrap();     _asm{ int 3 }
```

You may need to play with the exact syntax to get it to compile. The stack code will generally continue executing after a **dtrap()**, but the **dtrap()**s usually indicate that something is wrong with the port. **NO PRODUCT BASED ON THIS CODE SHOULD BE SHIPPED UNTIL THE CAUSES OF ALL CALLS TO dtrap() HAVE BEEN ELIMINATED!** When it comes time to ship code, the **dtrap()**s can be redefined to a null function to slightly reduce code size.

The next few primitives have the same function and syntax as **printf()**. They have separate names so that they can have their output redirected or be completely disabled independently of each other. The first, **dprintf()**, is used throughout the stack code to print warning messages when something seems to be wrong. This should be mapped to a debugging console or log during development, and generally **ifdef**ed away for FCS. The **ns_printf()** call is for printing statistical information from the DHCP “menus” functions. These will certainly be useful during product development, and depending on the nature of the product may be needed in the end user's release.

In most ports, these can both be mapped to **printf()** as shown while the product is under development. Note: This example works on Microsoft C, but some compilers will complain about this syntax since it ignores the fact that these names have parameters. You may have to experiment.

```
#define ns_printf   printf   /* same parms as printf, but works at init time */
#define dprintf     printf   /* same parms as printf, but works during run time */
```

For some products, it may make sense to define these away before FCS.

```
#define ns_printf(…)         /* define to nothing */
#define dprintf      (…)     /* define to nothing */
```

The last debugging tool in **dhcpport.h** is the **#define NPDEBUG**. Defining this will cause the debug code to be compiled into the build. This code does things like check for valid parameters and sensible configurations during runtime. It frequently invokes **dtrap()** or **dprintf()** to inform the programmer of detected problems. You will want to make sure it is defined during development. Unless PROM space is tight, it is OK to leave it defined for FCS - there will be no noticeable performance hit from this code.

```
#define NPDEBUG        1       /* enable debug checks */
```

2.3.5 Features and Options

The DHCP server can optionally read a standard UNIX-like **bootptab** file and use it to initialize entries in the DHCP address pool. This is provided for backward compatibility with BOOTP. This feature requires access to a conventional file system via the standard C calls **fopen()**, **fgets()** and **fclose()**. If you are designing a DHCP server for use with older UNIX or Windows 3.1 workstations, you should try to include this feature. To do this, **dhcpport.h** should contain this definition:

```
#define BOOTPTAB 1 /* will try to read a BSD bootptab file */
```

DHCP service requires one or more devices for sending and receiving network packets. These are usually hardware devices such as Ethernet or PPP ports. Most DHCP servers on embedded systems support only one device, (most commonly Ethernet), however devices like routers may need two or more. The InterNiche stack supports multiple logical devices, and has been used with up to eight. The structures to manage these devices are statically allocated at compile time, so the maximum number of devices the system will use at runtime must be set with a **#define**. If you are using the InterNiche IP stack, the define is already set in **ipport.h**, else you must define it in **dhcpport.h**.

```
/* define the maximum number of hardware interfaces */
#define MAXNETS 2                          /* max entries of nets[] array */
```

Finally, there are a few data field upper limits you will need to set:

```
#define CLIDSIZE            6         /* client ID size, usually ethernet address */
#define DHCPNAMESIZE 32        /* maximum name length */
```

2.4 dhcpport.c - The "glue" Layer

Once you've developed your **dhcpport.h** file as described in the previous section, the next step is to code the glue layers. These are the routines which map the generic service requests DHCP makes to specific services your target system provides. You may have already handled most of them through **#define** mapping in **dhcpport.h**. The rest need to be implemented as minimal layer of C code. In the demo packages most these are collected in the file **dhcpport.c**. You can name these files anything you like or implement these routines in multiple source files. This document presumes they are all in **dhcpport.c**.

2.4.1 UDP Hooks

Usually the most complex part of the glue layers is the network interface. DHCP need to send and receive packets via the UDP protocol. If you are using InterNiche's lightweight API or a standard "Sockets" interface, sample **dhcpport.c** files are provided which do most of the work for you. Otherwise, you will need to implement the routines described in the section starting on page 15. These are summarized below:

```
/* dhcp server initialization: */
int dhcp_init(void);
/* dhcp server's per-port utility for sending datagrams */
int dh_udp_send(int iface, void * outbuf, int outlen);

/* get the IP address associate with an iface number. This is
returned in net endian... */
ip_addr   dh_get_ip(int iface);
```

Experienced network programmers will realize the above routines do not provide a mechanism for DHCP to receive packets, only to initialize and send. Receiving is accomplished via a callback - a routine inside DHCP which is called from user code when a packet is received for the DHCP server. The form of the callback is:

```
/* portable dhcp server received packet handler */
int dhcp_receive(int iface, struct bootp * bp, unsigned len);
```

This callback is described with the other UDP hooks since the sense of the **iface** parameter is the same.

2.4.2 Timers and Multitasking

A DHCP server only needs to get CPU time upon two events, each of which is handled by a callback routine. The events are an arriving DHCP packet, and the once-per-second timer. The arriving DHCP packets are processed in **dhcp_receive()** which is described briefly above and detailed on page 31. The once-per-second timer is implemented by calling **dhcp_timeisup()** (details on page 32) once a second - the code to initialize this timer may be part of **dhcpport.c**.

The other aspect of multitasking is to protect sensitive structures from being corrupted by code re-entry. This is accomplished by two macros which protect critical sections of code:

```
#define ENTER_DHCP_SECTION();   {_asm{ pushf };  _asm{ cli } }
#define EXIT_DHCP_SECTION();   _asm{ popf };
```

The examples given are for the DOS port, where simple disabling interrupts for a brief period is sufficient. On a true real-time system, these should be mapped to a **mutex**.

Finally, a few bits of static data is required in **dhcpport.c**. The first set of these are the path/names where the database files are to be found. These can be hardcoded for some products or made user settable. The **name** strings are:

dhcpdef
dhcpdbase
dhcpsfile

An example setup is shown here:

```
/* names of DHCP database and configuration files */
char * dhcpdef = "dhcpsrv.nv";   /* Default DHCP values list */
char * dhcpdbase = "dhcpdb.nv";  /* client info (current leases) database */

/* DHCP dynamic database (disk file) for PC DOS demo port. This
should not be a relative path since the FTP server may CD us around. */
char * dhcpsfile = "\\etc\\dhcprecs.nv";
```

Note that all are **char*** and thus can be left **NULL** at compile time and assigned later.

Lastly, some static structures are expected which contain an initial free address pool and default IP information. The hardcoded values show below can be used in a configuration-less DHCP server which will be assigned IP information on a local “10-net”. This is designed to facilitate integration with InterNiche’s NAT Router product, making the setup of a factory floor or small local LAN totally independent of public IP addressing.

```
/* hardcoded "Factory" defaults for DHCP parameters values */
struct dh_range dhsys_range = {
  NULL,                 /* no link to next */
  0x00000002,      /* low address */
  0x007FFFFF,      /* high address */
};

/* Default values for DHCP assignments. All these are in LOCAL endian.
These are generally over-written from NVRAM entries. */

struct dh_ifdefaults dhsys_defaults = {
  0xFF000000,      /* subnet mask */
  0x0a000001L,     /* default assignment for default router (gateway) */
  0xCC9C8001L,           /* default Domain Name server */
```

```
   0xffff,            /* options mask, all options enabled */
   30,                /* default lease time, in seconds */
   "private",         /* domain name */
   1,                 /* net number, 0 thru (ifNumber-1) */
   100,               /* low-water mark for free address count */
};
```

These default options are generally overwritten by the values in the **dhcpsrv.nv** file, however one of these structure per net must be present in the build image.

2.5 The DHCP Database

2.5.1 Setting the Database Parameters

Before you can use your DHCP server, you will need to set up some basic IP address database information. This database contains the IP addresses, subnet masks, servers, lease times, names, etc., which will be assigned to your DHCP clients. The thread of control for doing this database setup moves across the user/dhcp code as follows:

1) User code (usually **dhcp_init()**) calls **dh_nvinit()** (part of **dhcpsnv.c**)
2) **dh_nvinit()**calls:
   a) **dh_nvread()** - which restores IP lease info from **nvram** (or file)
   b) **dh_parmset()** - which reads in option defaults
   c) **read_bootptab()** - (optional) reads in **bootptab** file

All this code is provided for systems which support a buffered file system (**fopen()**, etc.). Other systems will have to re-write the routines in steps 2a and 2b to extract the data (defined below) from their storage system.

2.5.2 Database Parameters List

The database parameters fall into two categories: those which are maintained on a per-client basis such a single permanent IP address, and those which are maintained per network, such as a pool of free IP addresses. When setting up the database, the end user will generally want to set up both types, for example if a web server obtains its IP address via DHCP, the end user can improve its accessibility by insuring that it always gets the same IP address from the DHCP server. On the other hand, a Windows 95 workstation whose only IP application is a Web browser can change IP addresses every time it is rebooted with no ill effects.

To facilitate managing this sort of data mix, the DHCP server maintains two types of data - one with detailed per-client information for permanent client assignments, another for “default” information for more generic client setup. Assignment of this data is done from a single file. In the demo port, this file is named **dhcpsrv.nv**, however this can be changed as part of the porting process. If the DHCP server is to work over multiple interfaces, one of these files must be provided for each interface.

The DHCP data items supported in the database are:

```
char name[DHCPNAMESIZE];  /* String for name */
ip_addr ipaddr;           /* client's assigned IP address */
ip_addr snmask;           /* client's assigned subnet mask */
ip_addr gwaddr;           /* client's assigned default gateway */
ip_addr dnsaddr;          /* Domain Name server */
char clientId[CLIDSIZE];  /* usually client's hardware address */
unshort   type;           /* type of this entry, see DHT_ defines below */
unshort   status;         /*status of this entry, see DHS_ defines below*/
long   tmo;               /* lease or offer timeout in cticks 0==infinite */
```

NOTE: In Intel x86 and other little-endian CPUs, the IP addresses (the **ip_addr** fields) are stored in **local** endian format, not net endian.

Most of these fields should be familiar to programmers with some exposure to TCP/IP networks, however the **clientId** field deserves some explanation. This field is the unique ID which the DHCP server will use to track each client as it asks for and receives an IP address and configuration. On old **BOOTP** systems this was always the Ethernet (or token ring) MAC address, since this was always unique. This tradition has generally been carried forward to DHCP, and if you are using DHCP over Ethernet or token ring, then using MAC address as client ID is by far the simplest way to go. The DHCP clients will determine the size of this field, and on these media it will always be six since both MAC addresses are six bytes in length. However since DHCP may be used over PPP and other address-less type links, there may be cases where the clientId field will not be six bytes in length. This is why **CLIDSIZE** is a **#define** and not hardcoded. If you implement a server which will use an unusual **clientId** size, be sure to modify the default value to the size your media will be using.

2.5.3 Suggested Database File Format

An issue you will encounter as you develop your DHCP server’s user interface is how to the end user will assign the database information. InterNiche provides the dhcpsnv.c source code file which will read in this information from the previously mentioned **dhcpsrv.nv** file. Since we expect this will be used in most DHCP server ports, the format of the **dhcpsrv.nv** file is described in the section. Remember, one of these files is needed for each network your DHCP server will serve.

The file is a plain text file which can be easily read and modified by a human operator. It is also designed to be easy to modify from a GUI front end, such a an embedded WEB server. Each data item occupies one line of text. The name of the data item comes first. A pattern match on this name is how **dhcpsrv.nv** identifies parameters, so the end user should understand that these must not be changed if he edits the file directly. Every item name ends with a colon character (“**:**”), the text after the colon is the data that **dhcpsnv.c** will attempt to convert into the correct data - usually an IP address, numeric parameter, or text string. If the **dhcpsnv.c** code detects syntax errors in the file, it will **dprintf** an error message and return an error code.

The first portion of the file covers the default database information. This is the setup information that will be given the DHCP clients unless overridden by a per **clientId** entry later one. The list of per **clientId** settings is the remainder of the **dhcpsrv.nv** file.

The fields of the default settings portion are as follows:

```
Default gateway: 10.0.0.1
Default DNS server: 204.156.128.1
Domain name: iniche.com
Default lease: 3600

Address Pool: 1
High address: 10.0.0.99
Low address: 10.0.0.2
Subnet mask: 255.0.0.0
```

All these match the fields of the same names described in the DHCP RFCs, and should be self explanatory to experienced TCP/IP programmers. None of these fields are actually required. Any that are missing in this section and also not specified in the per-client section which follows will simply not be assigned (or in the case of required IP addresses assigned as zero). If there are no addresses in the free address pool (and the requester's **clientId** is not included in the per-client list), no IP addresses will be handed out - the request will be silently ignored. If the lease time is omitted, it is assumed to be an infinite lease.

If discontiguous blocks of IP addresses are desired, more than one IP address free pool can be specified. In the above example, the format for a second address pool is:

```
Address Pool: 2
High address: 10.0.1.99
Low address: 10.0.1.2
Subnet mask: 255.0.0.0
```

This pattern can be continued indefinitely.

The per-client portion of the file is formatted as follows:

```
Client ID: 00006090068b
Host name: Felix the Cat
lease time: 3600
IP address: 10.0.0.33
subnet mask: 0.0.0.0
gateway: 10.0.0.1
dns server: 0.0.0.0
```

This follows the same rules as the previous section, however any omitted parameters (except host name) will use the defaults. Omitting the host name will result in no host name being offered to the client. Clients which request a particular host name (such as Windows 95 DHCP clients) will be allowed the requested host name unless it conflicts with another DHCP client already know to the DHCP server.

2.6 Testing

Once your **dhcpport.h** file is set up and your glue layers are coded, compiled, and linked, you are ready to test your server. The steps for a basic test are simple: start your DHCP server, then reboot any DHCP client machine. The two machines should complete a 4 packet exchange as described in the DHCP RFCs. If you have replaced the **dhcpsnv.c** file IO code with your own, then you should also test to ensure that both per-client data and host data are being set properly.

In any case you should now have a working DHCP server. Hopefully your porting was fast, easy, and fun. If you have any suggestions as to how this manual or the InterNiche DHCP server could be easier to understand or port, please contact us with them. Contact information is at the beginning of this manual.

3. TROUBLESHOOTING

If the event your implementation of the DHCP server has problems, there are several techniques you can use to track down the problems. The problems will generally fall into two categories: connecting the server to UDP and keeping the database information accurate.

3.1 UDP Transport

Although InterNiche provides code for most common UDP APIs, there are still a variety of things which can go wrong. Since the DHCP server always operates by responding to client requests, the first problem you are likely to encounter is the inability to receive packets. If you have tried rebooting a DHCP client and the server has not responded, you will want to make sure the DHCP server actually received the packet from the client. The easiest way is to use the **dhsrv** command in the DHCP server menus - it provides counters for all types of packets received and sent. If these are all zero, this tells you to go back and debug your UDP "listen" and receive code.

If the menu counters indicate a discover packet was received and an offer packet was sent, but no request was received, it is possible your UDP send has problems - the server thinks it sent the packet to the UDP layer, but UDP never got it onto the network. Debugging your **dh_udp_send()** code is then indicated.

The DHCP server, unlike many networking protocols, is quite amenable to source level debugging with breakpoints. Since each DHCP packet is sent from the server as a reply to a client packet, breakpoint setting a breakpoint on **dhcp_receive()** will allow you to trace the entire DHCP transaction all the way through to the sending of the response.

In all cases, a Packet Analyzer is an invaluable tool for debugging this sort of problem. These are available as software programs for Windows 95, or as dedicated hardware devices. An analyzer will capture on packets on the LAN to which it is attached, and save them for later review. Most support filters, so you can set them to capture only the packets of interest - in this case BOOTP/DHCP packets. Older analyzers may only filter at a coarser level, such as all IP packets, or all UDP packets. Older analyzers will also treat DHCP packets as BOOTP packets.

3.2 Database Debugging

In the event that the DHCP packets are being exchanged between client and server, but the IP configuration information is not what you expected. There are some simple techniques you can use to run down the problem.

First, of course double check you database files. Mistyped MAC addresses are a common source of trouble in per-client setups. The clients will not be found in the database and will be assigned default values instead.

The next step is to make sure the files are being read into the DHCP server's internal structures correctly. The menu system's **dhlist** and **dhentry** commands can be used to display information even for clients which have not generated a request yet. If the IP configuration information is not correct here, it will not be correct on the net.

Next step is top use a packet analyzer to check the information in the reply packets coming out of the server. If the packets do not reflect the data revealed by **dhentry** then there is an encoding problem of some kind. The most common cause of this is "endian" issues.

3.3 Windows 95 DHCP Client Bugs

Windows 95 workstations (at least those shipped through 1997) have several notable bugs and idiosyncrasies. The most obvious is that the DHCP **discover** and **request** packets are the old BOOTP size (about 346 bytes), instead of the larger DHCP size (about 500 bytes). The Windows 95 stations appear to work properly whether or not the reply is of DHCP or BOOTP size, however the InterNiche DHCP server tracks the size of the incoming packets and replies in kind. Hopefully this will keep us operating with future Microsoft releases whether or not they fix their bugs.

4. THE DHCP USER MENU

The DHCP server comes with portable C code to implement a few simple diagnostic commands on command line interface. The commands can be invaluable both during debugging of the server and to the end user during configuration and runtime. If you do not implement these menu commands as provided, we strongly suggest that some alternative method (i.e. a GUI) be provided to the end user for accessing the same data.

The menu commands are summarized below:

dhsrv - display DHCP server statistics
dhlist - list DHCP server assigned addresses
dhentry - list specific entry details
dhdeleted - delete a DHCP entry

The use and output of these commands is illustrated below. The first is **dhsrv**, which displays packet statistics for the server.

```
INET> dhsrv
plain bootp requests received: 0
plain bootp replys sent: 0
discover packets received: 0
offer packets sent: 0
dhcp request packets received: 0
declines received: 0
releases received: 0
acks sent: 0
naks sent: 0
requests for other servers: 0
protocol errors; all types: 0
```

All these packet types are described in the DHCP RFCs. Note that plain BOOTP packets are kept in separate categories.

The next command is **dhlist**. This lists a summary of all the database entries for know DHCP clients.

```
INET> dhlist
 1 IP:10.0.0.34 - client ID:00:00:60:90:06:8C - status:Unassigned
 2 IP:10.0.0.2 - client ID:00:00:F4:90:10:52 - status:Assigned via DHCP
 3 IP:10.0.0.33 - client ID:00:00:F4:90:0E:D8 - status:Assigned via DHCP
 4 IP:10.0.0.3 - client ID:00:40:C8:04:63:FA - status:Assigned via BOOTP
4 Entries
```

Note that this list includes clients defined in the database files, but not yet assigned via the DHCP protocol. This list is an effective tool for spotting non-existent machines, perhaps due to a mistyped MAC address or device which has been retired from the net. The unassigned status can also simply mean the client's lease has not yet expired, or the machine is currently powered off. The first entry above (10.0.0.34) is an example of this.

In the case above, as on most networks, the **clientId**s are Ethernet addresses. The Ethernet addresses in lines 2) and 4) are apparently not in the per-client list, since they were assigned IP addresses from the free address pool.

The next command of interest is **dhentry**. This displays all the database items assigned (or which will be assigned) for this client. The example below is for the first (#1) entry from the **dhlist** command's output in the previous example.

```
INET> dhentry 1
IP:10.0.0.34 - client ID:00:00:60:90:06:8C - status:Unassigned
subnet:255.0.0.0 gateway:10.0.0.1 DNS:204.156.128.1
lease 0, type: dbase, name: Felix the Mouse
```

This data was taken from the **dhcpsrv.nv** file excerpts in the section beginning on page 18. Note that some of the parameters (e.g. the name) are taken from the per-client entry for this MAC address, others (e.g. the DNS server) are from the default values.

5. USER PROVIDED FUNCTIONS

The functions described in this section must be provided by the porting programmer as part of the porting the InterNiche DHCP server. The DOS demo package can be referenced for examples. In you are using the InterNiche IP Stack, many for these functions are already provided therein.

In the demo packages these functions are either mapped directly to system calls via MACROS in **dhcpport.h,** or they are implemented in **dhcpport.c**

5.1 General Functions

NAME

dtrap()

SYNTAX

void dtrap(void);

DESCRIPTION

This primitive is intended to hook a debugger whenever it is called.

See the detailed description in the debugging aids section starting on page 12.

RETURNS

Usually nothing, depends on user modifications.

NAME

dprintf()
ns_printf()

SYNTAX

```
void dprintf(char *, ...);
void ns_printf(char *, ...);
```

DESCRIPTION

These two routines are functionally the same as **printf**. Both are called by the stack code to inform the programmer or end user of system status. **dprintf()** prints error warnings during runtime, and **ns_printf()** is used by the menu routine to display state information.

See the detailed description in the Debugging Aids section starting on page 12.

NAME

ENTER_DHCP_SECTION()
EXIT_DHCP_SECTION()

SYNTAX

void ENTER_DHCP_SECTION(void);
void EXIT_DHCP_SECTION(void);

PARAMETERS

None

DESCRIPTION

These two primitives should be designed to be paired around sections of code that must not be interrupted or pre-empted Generally these simply need to disable and re-enable interrupts. On UNIX-like systems they can be mapped to the **spl()** primitive. On Windows DLLs they can be defined to NULL functions since Windows message based system always runs to completion. Examples for embedded Intel x86 processors are provided in the demo. Only the definitions are given here; for examples see the source code.

The stack source code always pairs these two in the same routines, the implementers can push values on the stack in **ENTER** and retrieve it in the following **EXIT**. The Intel x86 example takes advantage of this to push the existing flags register on the stack, saving the interrupt flag state, and retrieves the value for the flags register later, restoring the interrupt flag as it was before the **ENTER** call.

RETURNS

These return no meaningful value.

5.2 UDP Network API Layer

These three functions are those which allow DHCP to send/receive UDP datagrams. Implementations are provided for standard Sockets and InterNiche's lightweight UDP API. The first few are calls the DHCP server code makes to the API layer code, whereas **dhcp_receive()** and **dhcp_timeisup()** are DHCP server internal functions which needs to be called from the UDP glue layer whenever a DHCP server packet is received.

NAME

dhcp_init()

SYNTAX

int dhcp_init(void);

PARAMETERS

None

DESCRIPTION

This should set up the UDP layer for IO and make the call to **dh_nvinit()** to initialize the database. When finished, a non-negative integer index should be assigned to each network interface with which DHCP will be doing packet IO. This interface index is used for both database setup and directing transmitted DHCP packets to the correct interface.

RETURNS

Returns **0** if no error, else a negative error code, as returned from the sub-layers (UDP and database).

NAME

dh_udp_send()

SYNTAX

int dh_udp_send(int iface, void * outbuf, int outlen);

PARAMETERS

iface	**-**	the index for the interface the packet is to be sent on.
outbuf	-	the data buffer containing the UDP header
outlen	**-**	length of the **outbuf**, usually BOOTP or DHCP header structure size.

DESCRIPTION

Broadcast a UDP datagram on the interface indicated. Buffer with UDP data to send and a length are passed.

RETURNS

Returns **0** if OK, else nonzero error.

NAME

dh_get_ip()

SYNTAX

ip_addr dh_get_ip(int net);

PARAMETERS

int net - the index of the network interface for the desired IP

DESCRIPTION

Gets the IP address associated with an interface number.

RETURNS

Returns the IP address in network endian. Return value is undefined if interface's address is not set or **iface** index is invalid.

5.3 UDP Callback Function

NAME

dhcp_receive()

SYNTAX

int dhcp_receive(int iface, struct bootp * bp, unsigned len);

PARAMETERS

iface - index of the interface the packet was received on
bp - pointer to the start of the BOOTP/DHCP header (UDP data)
len - length of **bp** structure, for Windows 95 support.

DESCRIPTION

This is called from the per-port protocol stack hooks whenever the UDP listen to the DHCP/BOOTP server port has received a DHCP packet. Length and interface have already been checked.

RETURNS

This routine returns **0** if OK, **-1** if packet has an error.

5.4 Timer Callback Function

NAME

dhcp_timeisup()

SYNTAX

void dhcp_timeisup(void);

PARAMETERS

None

DESCRIPTION

The DHCP clock tick. This should be called once a second by the host system. It allows the DHCP server to track lease time-outs and recycle unclaimed IP addresses.

Index