Porting the ARM DHCP Server

Version 1

Programmer's Guide

ARM

ARM DUI 0123A

Release Information

The following changes have been made to this document.

Change history

Date	Issue	Change
June 1999	A	First release

Proprietary Notice

Contents
Programmer's Guide

Preface

This preface introduces the ARM DHCP Server porting procedure. It contains the following sections:

About this book

This book is provided with the ARM DHCP Server software.

It is assumed that the ARM DHCP Server porting sources are available as a reference. It is also assumed that you have access to programmer's guides for C and ARM assembly language.

Intended audience

This Programmer's Guide is written for experienced embedded systems programmers, with a general understanding of what a DHCP server does. It is written for those programmers who want to port the ARM DHCP Server to an embedded system.

Using this book

This book is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter for an introduction to the ARM *Dynamic Host Configuration Protocol* (DHCP) server porting procedure.

Chapter 2 *Overview and Requirements*

Read this chapter for an overview of DHCP and the system requirements for porting the ARM DHCP Server.

Chapter 3 *Porting Step-by-Step*

Read this chapter to learn what you need to do, step-by-step, to port the ARM DHCP Server to an embedded system.

Chapter 4 *Troubleshooting*

Read this chapter for a description of some common problems which could arise when porting the ARM DHCP Server, and to learn methods for tracking and fixing them.

Chapter 5 *User-provided Functions*

Read this chapter for a description of the functions and primitives that you must provide as part of the porting process.

Typographical conventions

The following typographical conventions are used in this document:

bold	Highlights interface elements, such as menu names. Also used for emphasis in descriptive lists, where appropriate.
italic	Highlights special terminology, denotes internal cross-references, and citations.
`typewriter`	Denotes text that may be entered at the keyboard, such as commands, file and program names, and source code.
`type`writer	Denotes a permitted abbreviation for a command or option. The underlined text may be entered instead of the full command or option name.
`typewriter italic`	Denotes arguments to commands and functions where the argument is to be replaced by a specific value.
`typewriter bold`	Denotes language keywords when used outside example code.

Further reading

This section lists publications from both ARM Limited and third parties that provide additional information on developing for the ARM DHCP Server.

ARM periodically provides updates and corrections to its documentation. See `http://www.arm.com` for current errata sheets and addenda.

See also the ARM Frequently Asked Questions list at:
`http://www.arm.com/DevSupp/Sales+Support/faq.html`

ARM publications

This book contains reference information that is specific to the ARM DHCP Server. For additional information, refer to the following ARM publications:

- *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041)
- *ARM Software Development Toolkit User Guide* (ARM DUI 0040)
- *Porting TCP/IP Programmer's Guide* (ARM DUI 0079).

Other publications

For other reference information that may be helpful in understanding the DHCP server, please refer to the following:

- Kernighan, Brian W. and Ritchie, Dennis M., *The C Programming Language*, 2nd Edition, 1988, Prentice-Hall (ISBN 0-13-110370-8)
- RFC 1700, Postel, J., Reynolds, J., "ASSIGNED NUMBERS", October 1994
- RFC 2131, Droms, R., "Dynamic Host Configuration Protocol", March 1997.

Feedback

ARM Limited welcomes feedback on both the ARM DHCP Server, and its documentation.

Feedback on the ARM DHCP Server

If you have any problems with the ARM DHCP Server, please contact your supplier. To help us provide a rapid and useful response, please give:

- details of the release you are using
- details of the platform you are running on, such as the hardware platform, operating system type and version
- a small standalone sample of code that reproduces the problem
- a clear explanation of what you expected to happen, and what actually happened
- the commands you used, including any command-line options
- sample output illustrating the problem.

Feedback on this book

If you have any comments on this book, please send email to `errata@arm.com` giving:

- the document title
- the document number
- the page number(s) to which you comments apply
- a concise explanation of your comments.

General suggestions for additions and improvements are also welcome.

Chapter 1
Introduction

This chapter introduces the ARM *Dynamic Host Configuration Protocol* (DHCP) Server porting procedure.

This chapter contains the following sections:

1.1 About the ARM DHCP Server

This Programmer's Guide is provided with the ARM portable *Dynamic Host Configuration Protocol* (DHCP) server sources. The purpose of this document is to provide enough information to enable a moderately experienced C programmer, with a reasonable understanding of DHCP, to port the ARM DHCP Server to a new environment.

It is assumed that the ARM DHCP Server demonstration program is available as a reference.

The ARM DHCP Server can be readily ported to any system that supports network communication by way of *User Datagram Protocol* (UDP). An example implementation is provided that uses the lightweight UDP interface of the ARM *Internet Protocol* (IP) stack.

1.2 Terms and conventions

In this document, the following terms are used:

client When used without other qualification, means the ARM DHCP client that is ported to an embedded system.

end user Refers to the person who ultimately uses your product.

packet A sequence of bytes sent on network hardware, also known as a *frame* or *datagram*.

server When used without other qualification, means the ARM DHCP Server as ported to an embedded system.

stack Means the TCP/IP and related code, as ported to an embedded system.

system Refers to the embedded system.

you Used to indicate the user or engineer who is porting the server.

Conventions used throughout the document, such as the use of bold or italic font, are explained in *Typographical conventions* in the Preface.

Chapter 2
Overview and Requirements

This chapter gives an overview of DHCP and the system requirements for porting the ARM DHCP Server. It contains the following sections:

2.1 Purpose of DHCP

DHCP 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. Soft configuration information includes:

- IP address
- name
- gateway
- default servers.

There are about 50 of these information items which can be assigned with DHCP, and DHCP is designed so that customized configuration items can be easily added.

DHCP is a client/server protocol, meaning that the machine with the DHCP database serves requests from DHCP clients. The clients typically initiate the transaction by requesting an IP address and, possibly, other information from the server. The server looks up the client in its database, usually using 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 *Domain Name System* (DNS) server, and routing information.

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

2.2 Overview of BOOTP

In this manual and other DHCP literature, you will find numerous cross-references to *Bootstrap Protocol* (BOOTP), which preceded DHCP. DHCP is a superset of BOOTP. The main differences between the two protocols are:

- the lease concept, which was created for DHCP
- DHCP has the ability to assign addresses from a pool.

The ARM DHCP client (sold as part of the ARM TCP/IP stack) can work with older BOOTP servers.

2.3 What is a port?

Figure 2-1 shows a simplified diagram of the events which drive a typical DHCP server. The vast majority of activity is initiated by DHCP packets from the clients. The server code also requires a periodic clock tick so it can detect non-network-driven events, such as a lease expiring.

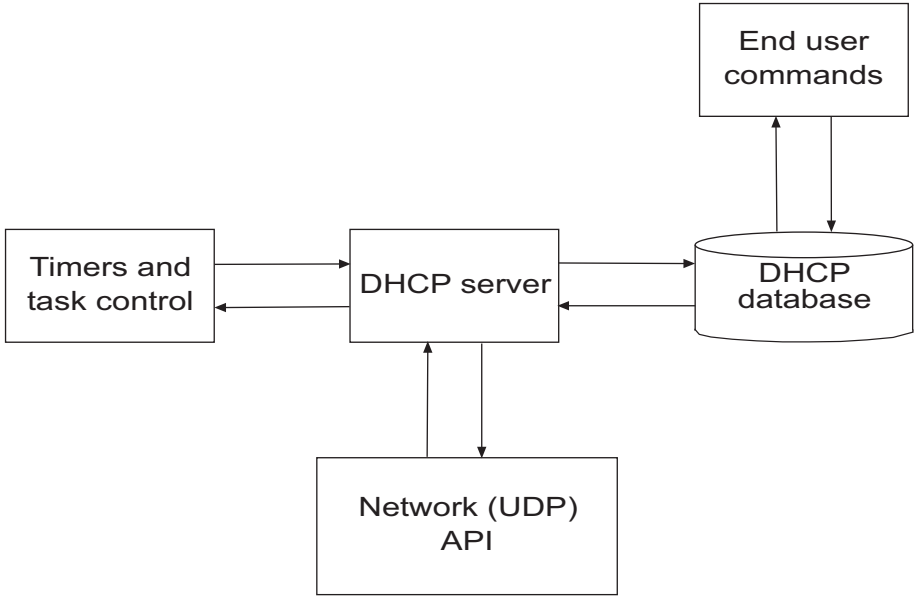

Figure 2-1 Events driving a DHCP server

While processing DHCP requests, the server will consult a local database. This database is generally set up by an end user. It contains the IP address pools, default information for clients, and, possibly, special configurations for specific clients. This database may be stored in a disk file or in nonvolatile memory. On some embedded systems, such as *Network Address Translation* (NAT) routers, it is recommended that you ship the server with a factory-configured database.

The stack designer does not know what tasking system, user applications, or interfaces will be supported in the target system. Therefore, 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. For example, when sending a packet, the stack would be designed with a generic `send_packet()` call, and you 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 ARM DHCP Server have evolved through years of porting to a variety of processors, network media, and tasking systems. Wherever possible, ARM has used standard interfaces (such as the ANSI C library) or included glue routines to illustrate their use.

The bulk of the work in porting a stack involves understanding and implementing these glue routines. The ARM DHCP Server has two kinds of glue routines:

- database access
- network access.

The calls to set and extract database items are *abstracted out*. This means the calls are generic, and will need to be provided as part of the port. Files are provided which use standard file input/output calls (such as `fopen()` and `fread()`) 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 provided. If your database will be kept in nonvolatile memory, and it does not have a file-system like API, you will need to develop your own data structures and the code to access them.

The other set of glue routines needed for a port is the network access set. It is recommended that you use the ARM DHCP Server with the ARM IP stack, because a file is provided which implements the required functions on the lightweight UDP API. Otherwise, you will need to provide some simple routines to allow the server to access your UDP protocol.

2.4 System requirements

Before beginning a port, you should ensure the necessary resources are available in the target environment. The ARM DHCP Server requires the following services from the system:

- access to a UDP layer with *listen* capabilities (such as the ARM lightweight API or sockets)
- a timer which ticks at least once per second
- a nonvolatile read/write method for storing database items (such as disk or nonvolatile memory)
- memory, as described in the next section, *Memory requirements*.

2.4.1 Memory requirements

It is difficult to provide exact memory requirements for the ARM DHCP Server. However, an estimate can be obtained by examining the results provided in the examples. The values in Table 2-1 are taken from the Menus example of the ARM TCP/IP stack, configured to include the server. The code was compiled for the ARM7TDMI processor core with version 2.50 of the ARM *Software Development Toolkit* (SDT), and was optimized for code size rather than speed. The figures do not include the UDP stack, generic menu drivers, or Standard C library functions, which would be required for full functionality.

Note

Because the code is subject to continuous development, the values in Table 2-1 may change with subsequent releases.

Table 2-1 ARM7TDMI memory requirements (in bytes)

	Built as ARM code	Built as Thumb code	Read-only data	Read-write data	Total data
DHCP server core	5304	3724	88	128	216
DHCP file access code	2652	2056	160	264	424
Menu front-end routines	464	388	72	72	144
UDP glue layer	436	316	72	72	144
Totals	8856	6484	392	536	928

2.4.2 Operating system requirements

The ARM DHCP Server also requires two services from the operating system:

clock tick The DHCP server needs to be called one time per second to free resources for addresses which have timed out.

dynamic memory allocation

The standard `calloc()` and `free()` library calls are ideal. However, the DHCP server can also use any other system, such as a partition-based system, with very little effort.

Chapter 3
Porting Step-by-Step

This chapter outlines what you need to do, step-by-step, to port the ARM DHCP Server to an embedded system.

This chapter contains the following sections:

3.1 Setting up your source tree

Before beginning the porting procedure (see *Porting procedure* on page 3-3), you should be aware of those files in the ARM distribution that are the *portable* files, and those that 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.

This section contains a list of ARM DHCP Server source files which should *not* need to be modified during a normal port. If you feel you need to modify one of these files during a routine port, please contact ARM technical support staff first, who will either suggest an alternative, or modify ARM sources to reflect the change.

The portable ARM DHCP Server source files, which should not require modification during a standard port, are:

`dhcpsrv\dhcpsrv.c`
: Contains the main DHCP server code.

`dhcpsrv\dhcpsrv.h`
: Contains internal declarations.

`dhcpsrv\dhcps.h`
: Contains internal declarations.

The database access file, which may need modification on some ports, is:

`dhcpsrv\dhcpsnv.c`
: Contains the file system database access code.

The ARM menu system routines can be used as delivered with the ARM menuing system. Otherwise, the routines may be replaced or omitted:

`dhcpsrv\dhcpmenu.c`
: Contains routines to dump statistics and the status.

The two per-port files usually need to be rewritten for the target system:

`dhcpsrv\dhcpport.c` **and** `dhcpsrv\dhcpport.h`
: Contain the glue layers for network access, memory allocation, time ticks, and other system-dependent calls.

3.2 Porting procedure

The list below describes the steps needed to port the ARM DHCP Server to a new environment. It is assumed that the server is being ported to a small or embedded system with a suitable API to access UDP operations, and that a minimal Standard C library, or equivalent, is available.

1. Copy the source files into your development environment (see *Setting up your source tree* on page 3-2).
2. If the target system does not support file system operations, write and compile the necessary code to load and modify the nonvolatile database (see *The DHCP database* on page 3-11).
3. Create a DHCP port file (see *The DHCP port file* on page 3-4) and compile the portable sources.
4. Write and compile the necessary code for the glue layer (see *Coding the glue layer* on page 3-8).
5. Build a target system image, linking with those libraries or objects that are required to perform UDP operations.
6. Test and debug the image (see *Testing* on page 3-17).

3.3 The DHCP port file

Before you compile the source files, you must create a version of the file `dhcpport.h`. This file contains most of the port-dependent definitions in the stack. CPU architectures (big-endian versus little-endian), compiler idiosyncrasies, and optional features (such as multiple interfaces and 10-net-only configuration) are controlled in this file.

Warning

A single mistake in this file (such as confusing big-endian with little-endian) may cause the port to work incorrectly. It is recommended that you spend time carefully implementing the file line-by-line to avoid any mistakes or typographical errors.

This section outlines the basic contents of `dhcpport.h`.

If the DHCP server is to be used with the ARM IP stack, the IP port file, `ipport.h`, should be included to ensure consistency. This file contains many of the definitions in this section.

3.3.1 Standard macros and definitions

The ARM DHCP Server expects `TRUE`, `FALSE`, and `NULL` to be defined within the scope of `dhcpport.h`. To do this, it is recommended that you 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
```

3.3.2 Memory allocation

The ARM 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:

```
#define DHE_ALLOC(size) calloc(1,size)
/* dhcp address list entry */
#define DHR_ALLOC(size) calloc(1,size)
/* address range structure */
#define DHD_ALLOC(size) calloc(1,size)
/* dhcp message structure */
```



```
#define DHE_FREE(ptr)   free(ptr)
#define DHR_FREE(ptr)   free(ptr)
#define DHD_FREE(ptr)   free(ptr)
```

Note

The `ALLOC()` macros must clear the allocated memory by setting it to zero.

Many *Real-Time Operating System*s (RTOS) do not use `calloc()` because of 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. Therefore, the macros can be mapped to a call to allocate the next largest partition size. The size requested for each macro is shown in Table 3-1.

Note

These sizes may change, depending on the compiler settings and the target system.

Table 3-1 Partition sizes

Macro	Size (in bytes)
`DHE_ALLOC()`	56
`DHR_ALLOC()`	16
`DHD_ALLOC()`	548

3.3.3 CPU architecture

Four common macros are used from Berkeley UNIX for doing byte-order conversions between different CPU architecture types. These are:

`htons()` converts 16-bit from local to network format

`htonl()` converts 32-bit from local to network format

`ntohs()` converts 16-bit from network to local format

`ntohl()` converts 32-bit from network to local format.

They may be either macros or functions. They accept 16-bit and 32-bit quantities as shown, and convert them from network format (big-endian) to the format of the local CPU. 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 these macros. The information below is given in case these macros are not already available.

For an ARM processor running in big-endian mode, these can simply return the variable passed, as in the following 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)
```

An ARM processor running in little-endian mode requires the byte order in 16-bit or 32-bit quantities to be swapped. The `lswap()` and `bswap()` primitives provided with the ARM demonstration program can be used, as follows:

```
#define htonl(long_var)   lswap(long_var)
#define htons(short_var)  ((u_short)(((u_short)(short_var)>>8) \
                                   | ((u_short)(short_var)<<8)))
#define ntohl(long_var)   lswap(long_var)
#define ntohs(short_var)  htons(short_var)
```

3.3.4 Debugging aids

The `dtrap()` primitive is called by the DHCP code whenever it detects a situation which should not occur. The purpose of the `dtrap()` primitive is to try to trap to whichever debugger may be in use by the programmer. It functions as an embedded break point.

The ARM example implements `dtrap()` as an empty function. A breakpoint can then be set on the function to alert a debug monitor such as Multi-ICE or the Angel Debug Monitor.

The stack code will generally continue executing after a call to `dtrap()`, but `dtrap()` usually indicates that something is wrong with the port.

Warning

You should not ship any product based on this code until all calls to `dtrap()` have been eliminated. When it is time to ship code, the `dtrap()` primitive can be redefined to a null macro to slightly reduce code size.

The last debugging tool in `dhcpport.h` is the `#define NPDEBUG`. This can be defined to cause the debug code to be compiled into the build. This code performs operations such as checking for valid parameters and sensible configurations during runtime. It frequently invokes `dtrap()` or `dprintf()` (see *dprintf() and ns_printf()* on page 5-3) to inform you of detected problems. You should ensure it is defined during development. Unless ROM space is low, it may be acceptable to leave it defined for release. It can be undefined for release to reduce code size by a small amount:

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

3.3.5 Features and options

The ARM 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 by way of 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, it is recommended that you 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 (Ethernet, most commonly), but some servers, such as routers, may need two or more devices. The ARM 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 ARM IP stack, this is already defined in `ipport.h` as `MAXNETS`:

```
/* number of interfaces on which we may support DHCP *
#define DHCPNETS MAXNETS
```

Finally, you will need to set upper limits for the sizes of certain data fields:

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

3.4 Coding the glue layer

After you have developed your `dhcpport.h` file, as described in the previous section, you must then code the glue layers. These are the routines which map the generic service requests DHCP makes to specific services provided by your target system. You may have already handled some of them through `#define` mapping in `dhcpport.h`. The rest need to be implemented as a minimal layer of C code. In the demonstration program, most of these are collected in the file `dhcpport.c`. You can name this file anything you want, or implement these routines in multiple source files. For the purposes of this documentation, it is assumed they are all in `dhcpport.c`.

3.4.1 UDP hooks

Usually the most complex part of the glue layer is the network interface. DHCP needs to send and receive packets by way of the UDP protocol. If you are using ARM's lightweight API, a sample `dhcpport.c` file is provided which does most of the work for you. Otherwise, you will need to implement the routines described in *UDP network API layer* on page 5-5. These are summarized below:

```
/* 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 network endian */
ip_addr   dh_get_ip(int iface);
```

The above routines do not provide a mechanism for DHCP to receive packets. Receiving is accomplished by way of a callback, a routine inside DHCP which is called from user code when a packet is received for the ARM DHCP Server. The form of the callback is:

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

3.4.2 Timers and multitasking

A DHCP server only needs to service two events, each of which is handled by a callback routine:

- an arriving DHCP packet
- the once-per-second timer.

The arriving DHCP packets are processed in `dhcp_receive()`, discussed briefly in *UDP hooks* above, and detailed in *dhcp_receive()* on page 5-7. The once-per-second timer is implemented by calling `dhcp_timeisup()` (see *Timer callback function* on page 5-8) once per second.

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

```
#define ENTER_DHCP_SECTION()
#define EXIT_DHCP_SECTION()
```

If you are using the ARM IP stack, these could be mapped to the `CRIT_SECTION()` or `NET_RESOURCE()` macros provided. For some applications, it may be acceptable to simply disable interrupts for a brief period. On a true real-time system, these might be mapped to a mutual-exclusion semaphore.

3.4.3 Database location and default values

A small amount of static data is required in `dhcpport.c`. The first set of data is the file paths of the database files. These can be hardcoded for some products or made to be user-configurable. The name strings are:

```
char  *dhcpdef;
char  *dhcpsfile;
```

An example setup is shown here:

Example 3-1

```
/* names of DHCP database and configuration files */
char * dhcpdef = "dhcpsrv.nv";   /* Default DHCP values list */

/* DHCP dynamic database (disk file) for demo port. This must be
an absolute path since the user application may change the
current working directory. */
char * dhcpsfile = "\\etc\\dhcprecs.nv";
```

These are pointers to strings and can therefore be left null at compile time and assigned at runtime.

3.4.4 Initialization

The following steps are required to initialize the server. In the ARM implementation, this is performed by the function `dhcp_init()` in `dhcpport.c`:

1. Set up the UDP layer to:
   - listen for incoming packets on the *DHCP server* port (port 67)
   - enable it to send packets.
2. Initialize the database (see *The DHCP database* on page 3-11). This can be achieved by a call to `dh_nvinit()` if the ARM-provided database access code is used. Otherwise, you should provide code to initialize the database, either from nonvolatile storage or from hard-coded defaults. The ARM example can be configured to use default configuration values and address ranges suitable for assigning information on a local 10-net. This is designed to facilitate integration with, for example, a NAT router product, making the setup of a network of factory-floor monitors, or small LAN, completely independent of public IP addressing. These default values will be overwritten by any values in the nonvolatile database.

3.5 The DHCP database

Before the ARM DHCP Server can be used, some basic IP address information is required. This is stored in a database containing information which will be assigned to DHCP clients. This information may include:

- IP addresses
- subnet masks
- servers
- lease times
- names.

The database is stored in memory, but most applications will require a nonvolatile copy of the database.

The ARM implementation includes routines to access the database, and to load default settings from a file (see *Suggested database file format* on page 3-12). You should initialize this by calling the function `dh_nvinit()`, as described in *dh_nvinit()* on page 3-15.

3.5.1 Database parameters

The database parameters are divided into two categories:

- those which are maintained on a per-client basis, such as a single permanent IP address
- 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 webserver obtains its IP address by way of DHCP, the end user can improve its accessibility by ensuring that it always obtains the same IP address from the ARM DHCP Server. However, a Windows 95 workstation whose only IP application is a Web browser can change IP addresses every time it is rebooted, while still allowing repeated access to the network.

To facilitate managing this type of data mix, the DHCP server maintains two types of data:

- That which contains detailed per-client information for permanent client assignments.
- That which contains *default* information for more generic client setup. Assignment of this data is done from a single file. In the demonstration program, this file is named `dhcpsrv.nv`, but this can be changed during 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 */
unsigned short   type;
/* type of this entry */
unsigned short   status;
/* status of this entry */
unsigned long    lease;
/* default lease duration, 0xffffffff==infinite */
```

Note

The IP addresses (the `ip_addr` fields) are stored in local-endian format, not network-endian.

Most of these fields should be familiar to programmers with some exposure to TCP/IP networks, but the `clientId` field requires elaboration. This field is the unique ID which the ARM 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 because this was always unique. This identification method has been adopted by DHCP, so if you are using DHCP over Ethernet or Token Ring, the simplest method is to use the MAC address as `clientId`. The DHCP clients will determine the size of this field, and on these media, the size will always be six because both MAC addresses are six bytes in length. However, because it may be used over PPP and other address-less links, there may be cases where the `clientId` field will not be six bytes in length. If you implement a server which will use a nonstandard `clientId` size, be sure to modify the definition of `CLIDSIZE` to the size your media will be using.

3.5.2 Suggested database file format

As you develop your DHCP server's user interface, you should consider the method the end user will use to assign the database information. ARM provides the `dhcpsnv.c` source code file which will read in this information from a file at runtime. Because this is expected to be used in most DHCP server ports, the format of the database file is described in this section.

Note

One of these database 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 easily modified from a GUI front-end, such as an embedded webserver. Each data item occupies one line of text. The name of the data item is first. The parameters are identified by performing a pattern match on this name, so the end user should understand that these must not be changed if editing the file directly. Every item name ends with a colon character (:). The text after the colon is the data which is usually an IP address, numeric parameter, or text string. If the database initialization function `nv_init()` 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 to the DHCP clients, unless overridden by a per-client entry at a later time. The list of per-client settings comprises the remainder of the database file.

Database file default settings fields

The fields of the default-settings portion are as follows:

```
   Net interface: 0
Default gateway: 10.0.0.1
Default DNS server: 204.156.128.1
Domain name: arm.com
Default lease: 3600
Default subnet: 255.0.0.0

Address Pool: 1
High address: 10.0.0.99
Low address: 10.0.0.2
Interface: 0
```

All these should be self-explanatory to experienced TCP/IP programmers. None of these fields are mandatory. Any that are missing in this section, and not specified in the per-client section which follows, will simply be assigned default values as set up in the DHCP initialization code. If there are no addresses in the free address pool, IP addresses will only be assigned to clients with an entry in the per-client list. Any requests will be ignored in this case.

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
Interface: 1
```

This pattern can be continued indefinitely.

Database file per-client fields

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

```
Client ID: 00006090068b
Host name: rose
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
```

The 12-digit `Client ID` field is a hexadecimal number representing the six bytes of the MAC address of this client.

This follows the same rules as those described in *Database file default settings fields* on page 3-13. However, any omitted parameters (except host name) will use the defaults. If you omit the host name, no host name will be 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 known to the DHCP server.

3.6 Nonvolatile storage functions

The functions in this section require an underlying nonvolatile storage system that can be accessed by way of a file system-like API, that is, one with standard functions such as `fopen()` and `fwrite()`. If such a system is not available, or if nonvolatile storage is not required, you will have to provide versions of `dh_nvadd()` and `dh_nvdel()`, both described below. It is permissible for them to do nothing. In this case, they could simply be defined to null macros in `dhcpport.h`.

Additionally, the database should be set up when the DHCP server is initialized, either from previously saved values or from default values. In the ARM example, `dh_nvinit()` shows how this can be done from a UNIX-like `bootptab` file, from the text format defined in *Suggested database file format* on page 3-12, or from a binary format used by the example `dh_nvadd()` and `dh_nvdel()` functions. Reference should be made to the example implementation (in particular, `dhcpsnv.c` and `dhcpsrv.h`) for details of the data structures.

3.6.1 dh_nvinit()

This function is provided with the ARM example database implementation and is called when DHCP is initialized to set up initial values for the database. It will read default values and initial per-client settings from the file given by `dhcpdef` (see *Suggested database file format* on page 3-12) and (optionally) from a UNIX-like `bootptab` file. It will then load any additional per-client entries which were stored by `dh_nvadd()` in a previous session.

Syntax

```
int dh_nvinit(void)
```

Return value

Returns one of the following:

0 If successful.

-1 If not successful.

3.6.2 dh_nvadd()

This function should store the data in the given list entry in nonvolatile storage, or update the entry if data from this list entry has already been stored. The `clientId` field of the list-entry structure uniquely identifies the list entry. In the ARM example implementation, this is written to the file given by `dhcpsfile` in a custom binary format.

Syntax

```
void dh_nvadd(struct dhcpent *dhp)
```

where:

dhp is a pointer to a DHCP address list-entry structure.

Return value

None.

3.6.3 dh_nvdel()

This function should find the given list entry in nonvolatile storage and, if found, remove it. The `clientId` field of the list-entry structure uniquely identifies the list entry.

Syntax

```
void dh_nvdel(struct dhcpent *dhp)
```

where:

dhp is a pointer to a DHCP address list-entry structure.

Return value

None.

3.7 Testing

After your `dhcpport.h` file is set up and your glue layers are coded, compiled, and linked, the server can be tested. To perform a basic test:

1. Start your DHCP server.
2. Reboot any DHCP client machine. The two machines should complete a four-packet exchange, as described in RFC 2131.
3. If you have replaced the ARM example file access code with your own, you should also test to ensure that both per-client data and host data are being set properly.

You should now have a working DHCP server.

Chapter 4
Troubleshooting

This chapter describes some common problems which could arise when porting the ARM DHCP Server, and outlines methods for tracking and fixing them.

This chapter contains the following sections:

4.1 Overview of troubleshooting

If your implementation of the ARM DHCP Server develops problems, there are various techniques you can use to track them. These techniques may involve any of the following:

- *UDP transport* on page 4-3
- *Database debugging* on page 4-4
- *The DHCP user menu* on page 4-5.

Problems can arise as a result of either:

- connecting the server to UDP
- attempting to keep the database information accurate.

4.2 UDP transport

Because the ARM DHCP Server always operates by responding to client requests, the first problem you may encounter is a failure to receive packets. If you have tried rebooting a DHCP client and the server has not responded, you should ensure the DHCP server actually received the packet from the client. To do this, use the `dhsrv` command in the DHCP server menus. It provides counters for all types of packets, both received and sent. A counter showing all zeros indicates that no packets are being received, and the problem is likely to involve the UDP listen or 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. For example, the server may have sent the packet to the UDP layer, but UDP never relayed it to the network. In this case, the problem is likely to involve the `dh_udp_send()` code.

The ARM DHCP Server, unlike many networking protocols, accepts source level debugging with breakpoints. Because each DHCP packet is sent from the server as a reply to a client packet, you can set a breakpoint on `dhcp_receive()`, enabling you to trace the entire DHCP transaction up to the sending of the response.

In all cases, a packet analyzer is the suggested tool for debugging this type of problem. These are available as software programs for most major operating systems, or as dedicated hardware devices. An analyzer will capture packets on the LAN to which it is attached, and save them for later review. Most packet analyzers 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 may also treat DHCP packets as BOOTP packets.

4.3 Database debugging

If 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 discover the problem:

1. Double check your database files. Mistyped MAC addresses, or other client IDs, are a common source of trouble in per-client setups. That is, the clients will not be found in the database and will be assigned default values instead.

2. Ensure the files are being read correctly into the DHCP server's internal structures. The menu system's `dhlist` and `dhentry` commands can be used to display information, even for clients that have not yet generated a request. If the IP configuration information is not correct here, it will not be correct on the network.

3. 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`, there is an encoding problem. Specifying the wrong byte order is the most common cause of this.

4.4 The DHCP user menu

The ARM DHCP Server includes portable C code to implement a few simple diagnostic commands on the debugging terminal. These commands can be helpful during debugging of the server, and can also be helpful to the end user during configuration and runtime. If you do not implement these menu commands as provided, it is recommended that you provide the end user with some alternative method for accessing the same data. The menu commands are summarized in Table 4-1:

Table 4-1 Menu commands

Menu command	Description
`dhsrv`	Displays DHCP server statistics
`dhlist`	Lists DHCP server assigned addresses
`dhentry`	Lists specific entry details
`dhdelete`	Deletes a DHCP entry

The use and output of the these commands are illustrated in the examples below. Example 4-1 displays packet statistics for the server.

Example 4-1 dhsrv

```
INET> dhsrv
plain bootp requests received: 0
plain bootp replies 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 RFC 2131. Note that plain BOOTP packets are kept in separate categories.

The next command is `dhlist`. Example 4-2 shows a summary of all the database entries for known DHCP clients:

Example 4-2 dhlist

```
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

This list includes clients defined in the database files, but not yet assigned by way of DHCP.

This list is an effective tool for detecting non-existent machines. For example, with this list, a mistyped MAC address, or a device which has been retired from the network, can be easily identified. The `Unassigned` status can also indicate that the client's lease has expired, or the machine is currently powered off. The first entry above (`10.0.0.34`) is an example of this.

In Example 4-2, as on most networks, the client Ids are ethernet addresses. The ethernet addresses in lines 2 and 4 are apparently not in the per-client list because they were assigned IP addresses from the free address pool.

The next command, `dhentry`, displays all the database items assigned (or, those which will be assigned) for this client. Example 4-3 is for the first (1) entry from the `dhlist` command's output in Example 4-2.

Example 4-3 dhentry

```
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: rose
```

This data was taken from `dhcpsrv.nv` file excerpts in *Database file default settings fields* on page 3-13.

Note

Some of the parameters, such as `name`, are taken from the per-client entry for this MAC address. Others, such as the DNS server, are taken from the default values.

Chapter 5
User-provided Functions

The functions and primitives described in this section must be provided as part of the porting process. The ARM example port can be referenced for examples. Many of these functions are provided in the ARM IP stack.

In the demonstration program, these functions are either mapped directly to system calls by way of macros in `dhcpport.h`, or they are implemented in `dhcpport.c`.

This chapter contains the following sections:

5.1 General functions

The functions described in this section are used by the server to perform the platform-specific tasks of reporting information to the user and guarding critical data.

5.1.1 dtrap()

This primitive is intended to hook a debugger whenever it is called. For more details, see *Debugging aids* on page 3-6.

Syntax

```
void dtrap(void)
```

Return value

None.

5.1.2 dprintf() and ns_printf()

These two functions perform the same operation as `printf()`. That is, both are called by the stack code to inform the programmer or end user of the system status. They have separate names so they can have their output redirected, or be completely disabled, independently of each other. The `dprintf()` function 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 redefined to a null macro for release. The `ns_printf()` function is for printing statistical information from the DHCP *menus* functions. These may be utilized during product development and, depending on the nature of the product, may be needed in the final release.

Syntax

```
int dprintf(const char *format, …)
```

```
int ns_printf(void *vio, char *format, …)
```

where:

format is a format string like `printf()`.

vio is a generic input/output pointer.

Return value

Ignored.

5.1.3 ENTER_DHCP_SECTION() and EXIT_DHCP_SECTION()

These two primitives should be paired around sections of code that must not be interrupted or pre-empted. A simple implementation involves disabling and re-enabling interrupts. This may be unacceptable on a real-time system, where these functions could map to a *mutual exclusion* (mutex) semaphore. This type of mechanism is generally provided by an RTOS.

Syntax

```
void ENTER_DHCP_SECTION(void)
```

```
void EXIT_DHCP_SECTION(void)
```

Return value

None.

Usage

The stack source code always pairs these two in the same routines. The implementor can store state information, for example, by pushing it onto a stack during the call to `ENTER_DHCP_SECTION()`, and restore it during the call to `EXIT_DHCP_SECTION()`.

5.2 UDP network API layer

This layer comprises three functions which allow DHCP to send and receive UDP datagrams. Implementations are provided for ARM's lightweight UDP API. The first two are calls the DHCP server code makes to the glue layer code, whereas `dhcp_receive()` and `dhcp_timeisup()` are DHCP server internal functions which need to be called from the UDP glue layer.

Each function is passed a *network interface index* argument. This should be an integer between 0 and `DHCPNETS-1` (defined in `dhcpport.h`), which uniquely identifies a network interface.

5.2.1 dh_get_ip()

This function obtains the IP address associated with an interface number.

Syntax

```
ip_addr dh_get_ip(int net)
```

where:

net is the index of the network interface.

Return value

Returns the IP address of the interface in network-endian.

5.2.2 dh_udp_send()

This function broadcasts a UDP datagram on the network interface indicated. A buffer with UDP data to send is passed, including a length.

Syntax

```
int dh_udp_send(int net, void *outbuf, int outlen)
```

where:

net is the index for the network interface on which the packet is to be sent.

outbuf is the data buffer containing the DHCP message.

outlen is the length, in bytes, of the buffer, which is usually the BOOTP or DHCP message structure size.

Return value

Returns one of the following:

0 If successful.

non-zero error value
If unsuccessful.

5.3 UDP callback function

The UDP callback function `dhcp_receive()` should be called upon receipt of a DHCP packet.

5.3.1 dhcp_receive()

This function should be called by the protocol stack whenever a BOOTP or DHCP message is received by UDP on the BOOTP server port (port 67).

Syntax

```
int dhcp_receive(int net, struct bootp *bp, unsigned len)
```

where:

net is the index of the network interface on which the packet was received.

bp is a pointer to the start of the BOOTP/DHCP message.

len is the length, in bytes, of the *bp* structure.

Return value

Returns one of the following:

0 If successful.

–1 If the packet contains an error.

5.4 Timer callback function

The timer callback function `dhcp_timeisup()` is a DHCP server function that should be called once per second.

5.4.1 dhcp_timeisup()

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

Syntax

```
void dhcp_timeisup(void)
```

Return value

None.

Index

The items in this index are listed in alphabetical order, with symbols and numerics appearing at the end. The references given are to page numbers.

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

W