

Wind River[®] Network Stack for VxWorks[®] 6

PROGRAMMER'S GUIDE
Volume 2: Application Protocols

6.6

Copyright © 2007 Wind River Systems, Inc.

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means without the prior written permission of Wind River Systems, Inc.

Wind River, Tornado, and VxWorks are registered trademarks of Wind River Systems, Inc. The Wind River logo is a trademark of Wind River Systems, Inc. Any third-party trademarks referenced are the property of their respective owners. For further information regarding Wind River trademarks, please see:

<http://www.windriver.com/company/terms/trademark.html>

This product may include software licensed to Wind River by third parties. Relevant notices (if any) are provided in your product installation at the following location:

installDir\productName\3rd_party_licensor_notice.pdf.

Wind River may refer to third-party documentation by listing publications or providing links to third-party Web sites for informational purposes. Wind River accepts no responsibility for the information provided in such third-party documentation.

Corporate Headquarters

Wind River Systems, Inc.
500 Wind River Way
Alameda, CA 94501-1153
U.S.A.

toll free (U.S.): (800) 545-WIND
telephone: (510) 748-4100
facsimile: (510) 749-2010

For additional contact information, please visit the Wind River URL:

<http://www.windriver.com>

For information on how to contact Customer Support, please visit the following URL:

<http://www.windriver.com/support>

Contents

1	Overview	1
1.1	Introduction	1
1.2	Technology Overview	2
1.2.1	Product Overview	3
1.3	Additional Documentation	3
	Wind River Documentation	3
	Online Resources	4
	Books	4
2	Network Application Protocols	5
2.1	Introduction	5
2.2	Ping	6
2.2.1	Ping Commands	7
2.2.2	Ping6 commands	9
2.3	DNS	11
	Technology Overview	11
2.3.1	Component Overview	12
2.3.2	Conformance to Standards	13

2.3.3	Configuring and Building DNS	13
2.3.4	Using nslookup from the Command Interpreter	16
2.4	SNTP	17
2.4.1	Configuring and Building SNTP	18
2.5	FTP	23
2.5.1	Configuring and Building FTP	24
2.5.2	Using the FTP Client from the Command Interpreter	29
2.6	TFTP	31
2.6.1	Configuring and Building TFTP	32
2.7	RSH	34
2.7.1	Configuring and Building RSH	35
2.7.2	Enabling Access to an RSH User	35
2.8	RPC	36
2.8.1	Configuring and Building RPC	37
2.9	rlogin	37
2.9.1	Configuring and Building rlogin	37
2.10	Telnet	39
2.10.1	Configuring and Building Telnet	39
2.11	Creating a netDrv Device for RSH or FTP	41
3	Wind River DHCP and DHCPv6: Overview	45
3.1	Introduction	45
3.1.1	Architectural Overview: Client, Server, and Relay Agent	46
3.1.2	DHCPv6	46
3.1.3	Build Configuration Parameters and sysvars	46

4	Wind River DHCP: Server	49
4.1	Introduction	49
	Conformance to Standards	49
4.1.1	Server Overview	50
4.1.2	Server Components	51
4.2	Including the DHCP Server in a Build	52
4.3	Setting Up Addresses, Options, Subnets and Hosts	59
4.3.1	Configuring the Server with the ipdhcps_netconf_sysvar Array	59
4.3.2	Configuring the Server with Shell Commands	62
	DHCP Server Shell Commands	62
4.4	Implementing Hook Routines for Initialization and Shutdown	66
4.4.1	The ipdhcps_start_hook() Routine	66
4.4.2	The ipdhcps_stop_hook() Routine	70
4.4.3	DHCP Server API Routines	71
4.5	Setting Options in Shell Commands and API Routines	72
4.5.1	Using Standard DHCP Options in Shell Commands and APIs	72
4.5.2	Using Wind River-Specific Options in Shell Commands and APIs	80
5	Wind River DHCP: Relay Agent	83
5.1	Introduction	83
	Relay Agent Overview	83
	Conformance to Standards	84
5.2	Including the DHCP Relay Agent in a Build	85
5.3	Configuring the Relay Agent with the ipdhcpr_netconf_sysvar Array	87
5.4	Using Shell Commands	88
5.5	Implementing the ipdhcpr_start_hook() Routine	89

5.5.1	DHCP Relay Agent API Routines	91
6	Wind River DHCP: Client	93
6.1	Introduction	93
6.1.1	Conformance to Standards	94
6.2	Including the DHCP Client in a Build	95
6.3	Using Shell Commands	100
6.4	Implementing the <code>ipdhcpc_option_callback()</code> Routine	101
6.4.1	DHCP Options Not Initially Implemented in the Client	103
7	Wind River DHCPv6: Server and Relay Agent	107
7.1	Introduction	107
7.2	Assigning Client-specific Authentication Keys	110
8	Wind River DHCPv6: Client	113
8.1	Introduction	113
8.1.1	Configuring the DHCPv6 Client	114
8.1.2	Conformance to Standards	118
8.2	Including the DHCPv6 Client in a Build	120
8.3	Using Shell Commands	132
9	Creating Network Applications as RTPs	135
9.1	Introduction	135
9.2	Running Network Applications in RTPs	136
9.2.1	General Network/RTP Incompatibilities	137
9.3	Working with Application RTPs	137

9.3.1	Building an RTP ELF Object File for a Network Application	137
9.3.2	Launching an RTP	139
9.3.3	Identifying the RTP Constructor Routine in a Library	140
9.3.4	Shutting down an RTP Application	141
9.4	Using Socket Connections with RTPs	142
10	Internet and Local Domain Sockets	145
10.1	Introduction	145
10.2	Configuring VxWorks for Sockets	147
10.3	Using Sockets in VxWorks	150
	Communications Domains	151
	Socket Types	153
10.4	Working with Local Domain Sockets	154
10.5	Working with Internet Domain Sockets	156
10.5.1	Creating the Connection for Internet Domain Stream Sockets	164
10.5.2	Sending and Receiving Data Using Internet Domain Sockets	166
10.5.3	Closing or Shutting Down an Internet Domain Socket Connection .	167
10.5.4	Support Routines for Working with Internet Addresses	168
Index	171

1

Overview

- 1.1 Introduction 1
- 1.2 Technology Overview 2
- 1.3 Additional Documentation 3

1.1 Introduction

The Wind River Network Stack is a dual IPv4/IPv6 TCP/IP stack that is designed for use in modern, embedded real-time systems. It includes many services and protocols that you can use to build networking applications.

This is the third volume of the *Wind River Network Stack Programmer's Guide*. For information on the following topics, see the *Overview* chapter of the *Wind River Network Stack Programmer's Guide, Volume 1*:

- an overview of the Wind River Network Stack
- a list of features unique to Wind River platforms
- a guide to relevant additional documentation
- where to get the latest release information

1.2 Technology Overview

This manual documents the following technologies that are part of the Wind River Network Stack:

- **ping**

The *ping* utility tests whether a particular host is reachable on the network.

- **DNS**

Domain Name Server (DNS) is a distributed database that applications can use to translate host names to IP addresses and back.

- **SNTP**

The Simple Network Time Protocol (SNTP) synchronizes the clocks of computer systems over packet-switched, variable-latency data networks.

- **FTP**

File Transfer Protocol (FTP) enables the exchange of files over a TCP/IP network.

- **TFTP**

Trivial File Transfer Protocol (TFTP) is a simple protocol used to exchange files with a remote server.

- **RSH**

The remote shell (RSH) allows users to remotely log in to servers and execute commands.

- **RPC**

Remote Procedure Call (RPC) allows a process on one machine to call a procedure that is executed by another process on another machine.

- **rlogin**

rlogin allows remote shell access to and from a target.

- **telnet**

telnet provides command-line login sessions between hosts on a network.

- **DHCP**

The Dynamic Host Configuration Protocol (DHCP) automates the configuration of computers that use TCP/IP.

- **sockets**

Using sockets, you can send and receive data over an IP network, communicate with other processes, access IP multicasting functionality, and review and modify the routing tables.

This manual also describes how to create network applications as Real-Time Processes (RTPs).

1.2.1 Product Overview

The technologies listed above are components that may be included in or excluded from your project build, depending on its needs.

1.3 Additional Documentation

The following sections describe additional documentation about the technologies described in this book.

Wind River Documentation

The Wind River Network Stack is described in the three volumes of the *Wind River Network Stack Programmer's Guide*:

- Volume 1 has an overview with general information about the network stack, and describes the Network and Transport layers.
- Volume 2 (this volume) describes application-layer protocols and socket programming.
- Volume 3 describes interfaces, drivers, and the MUX, which is an abstraction layer between drivers and protocols or services.

The *Getting Started* guide for your Platform includes instructions on how to build a component or product into VxWorks, either through the Workbench Kernel Editor or the **vxprj** utility.

For information on using Workbench to create a VxWorks Image Project and to include build components, see the *Wind River Workbench User's Guide for VxWorks*.

For information on using the **vxprj** command-line utility, see the *VxWorks Command-Line Tools User's Guide*.

For more information on RTPs, and information on RTP projects in Workbench, see the *Wind River Workbench User's Guide* and the *VxWorks Programmer's Guide*.

For information on Transparent Inter-Process Communication (TIPC) domain sockets, see the *Wind River TIPC for VxWorks 6 Programmer's Guide*.

The *Wind River Platforms for VxWorks Migration Guide* details how to migrate from an earlier release of the network stack.

Online Resources

Online resources are as follows:

- The Internet Engineering Task Force, <http://www.ietf.org>

Books

Additional documentation is as follows:

- *Internetworking with TCP/IP, Volume I: Principles, Protocols, and Architecture*, Douglas E. Comer.
- *UNIX Network Programming, Volume 2, Second Edition* by W. Richard Stevens

2

Network Application Protocols

2.1	Introduction	5
2.2	Ping	6
2.3	DNS	11
2.4	SNTP	17
2.5	FTP	23
2.6	TFTP	31
2.7	RSH	34
2.8	RPC	36
2.9	rlogin	37
2.10	Telnet	39
2.11	Creating a netDrv Device for RSH or FTP	41

2.1 Introduction

This chapter describes the network application protocols in the Wind River Network Stack and the configuration components and parameters associated with each.

Including Application Protocols in Projects

The Getting Started guide for your Platform includes instructions on how to build a component or product into VxWorks, either through the Workbench Kernel Editor or the **vxprj** utility. The remainder of this chapter describes the various network application protocol components and their configuration parameters.

Dynamic Configuration of Application Protocol Components

You can configure network application protocol components through the Workbench Kernel Editor at build-time, or through the **sysvar** command at run-time. There are equivalent **sysvar** command variables for most configuration parameters. You can find the names of these variables in the configuration parameter tables in this book. For information on how to use the **sysvar** command, see *Wind River Network Stack for VxWorks 6 Programmer's Guide, Volume 1: Transport and Network Protocols*.

Shell Commands

Issue shell commands for the components described in this chapter from the VxWorks command interpreter. For information on including and using shell commands, see *Wind River Network Stack for VxWorks 6 Programmer's Guide, Volume 1: Transport and Network Protocols*.

2.2 Ping

The *ping* utility tests whether a particular host is reachable on the network.

Technology Overview

Ping sends ICMP echo requests (**ECHO_REQUEST** packets) to the target host and listens for replies (**ECHO_RESPONSE** packets). The Wind River Network Stack implements ping for both IPv4 and IPv6.

Component Overview

Wind River supplies a ping implementation that includes the **ping** and **ping6** commands and the API wrappers (for backward compatibility) that encapsulate the commands. You execute ping from the command line.

Configuring and Building Ping

The Wind River Network Stack has the following ping client configuration components:

- IPCOM ping commands (`INCLUDE_IPPING_CMD`)
- IPCOM ping6 commands (`INCLUDE_IPPING6_CMD`)
- PING client (`INCLUDE_PING`)
- PING6 client (`INCLUDE_PING6`)

PING Client and PING6 Client

The `INCLUDE_PING` and `INCLUDE_PING6` components pull in wrappers for the **ping** and **ping6** shell commands. These wrappers are for backward compatibility with an earlier release of the Wind River Network Stack, and you need not use them in new applications. For details on migrating from an earlier release of the network stack see the Migration Guide for your Platform.

2.2.1 Ping Commands

Include the `INCLUDE_IPPING_CMD` component in your project in order to enable the **ping** shell command.

Using Ping from the Command Interpreter

Syntax:

```
ping [-AbDnrx] [-c count | -t] [-I interfaceName] [-Q tos] [-s size] [-S sourceAddress] [-t ttl] [-v routeTable] [-w timeout] host
```

Description:

ping sends an ICMP `ECHO_REQUEST` datagram to elicit an ICMP `ECHO_RESPONSE` from a host or gateway. `ECHO_REQUEST` datagrams—pings—have an IP and an ICMP header, followed by a **struct timeval** and then an arbitrary number of pad bytes used to fill out the packet.

The command-line options are described in [Table 2-1](#).

Table 2-1 Ping Shell Command Options

Option	Description
-A	Add router alert option to each sent ping request.
-b	Allow the pinging of a broadcast address.
-c <i>count</i>	Stop after sending (and waiting the specified delay to receive) <i>count</i> number of ECHO_RESPONSE packets. Set <i>count</i> to -1 if you do not want the ping to end on its own.
-D	Set the "DF" (don't fragment) flag on each sent ping request.
-I <i>interfaceName</i>	Specify the outgoing interface.
-n	Use numeric output only. Ping will not attempt to look up names for host addresses.
-Q <i>tos</i>	Specify the Type of Service field in the IPv4 header.
-r	Bypass the normal routing tables and send ping requests directly to a host on an attached network, by setting the SO_DONTROUTE option..
-s <i>size</i>	Set the number of data bytes to be sent. The default is 56, which translates into 64 ICMP data bytes when combined with the 8 bytes of ICMP header data.
-S <i>sourceAddress</i>	Set the address to use as a source address when sending.
-t <i>ttl</i>	Set the time to live (TTL), in hops.
-V <i>routeTable</i>	Set the route table to the one specified by the index number <i>routeTable</i> . The default is 0 (zero). This option is only valid if you have enabled virtual routing.
-w <i>timeout</i>	Set the number of milliseconds between ping requests. The default is 1000 milliseconds.
-x	Send a ICMP timestamp request instead of a ECHO request.
<i>host</i>	The IPv4 address of the host to ping.

ping sends 4 datagrams by default, one per second, and prints one line of output for every **ECHO_REQUEST** returned. **ping** computes, and may display, round-trip times and packet loss statistics.

TTL

The TTL value of an IP packet represents the maximum number of “hops,” or routers, that the packet can go through before it is discarded. Typically each router will decrement the TTL field by exactly one when forwarding the packet and will discard the packet if its TTL field reaches zero.

The TCP/IP specification states that the TTL field for TCP packets should initially be set to 60, but many systems set it to smaller values (4.3BSD uses 30, 4.2BSD uses 15). The maximum value of this field is 255. Most UNIX systems set the TTL field of ICMP **ECHO_REQUEST** packets to 255.

Normally, **ping** outputs the TTL value from the packet it receives. When a remote system receives a ping packet, it can do one of three things with the TTL field in response:

- Leave it unchanged; this is what Berkeley UNIX systems did before the 4.3BSD-Tahoe release. In this case the TTL value in the packet will be 255 minus the number of routers in the round-trip path.
- Set it to 255; this is what current Berkeley UNIX systems and the network stack do. In this case the TTL value in the packet will be 255 minus the number of routers in the path from the remote system to the pinging host.
- Set it to some other value. Some machines use the same value for ICMP packets that they use for TCP packets, for example either 30 or 60.

2.2.2 Ping6 commands

Include the **INCLUDE_IPPING6_CMD** component in your project in order to enable the **ping6** shell command.

Using ping6 from the Command Interpreter

Syntax

```
ping6 [-c count | -t] [-F label] [-h limit] [-n] [-r] [-Q class] [-S sourceAddress] [-T] [-s packetSize]
[-v routeTable] [-w timeout] [hop ...] host
```

Description

ping6 sends the ICMPv6 **ICMP6_ECHO_REQUEST** datagram to elicit an **ICMP6_ECHO_REPLY** from a host or gateway. **ICMP6_ECHO_REQUEST** datagrams—pings—have an IPv6 header and an ICMPv6 header formatted as documented in RFC 2463.

The command-line options are described in [Table 2-2](#).

Table 2-2 **Ping6 Shell Command Options**

Option	Description
-c <i>count</i>	Stop after sending (and receiving) <i>count</i> number of ICMP6_ECHO_REPLY packets. Set <i>count</i> to -1 if you do not want ping6 to stop on its own (or use the “-t” option).
-F <i>label</i>	Specify the flow label in the IPv6 header.
-h <i>limit</i>	Set the IPv6 hop limit.
-n	Request numeric output only; ping6 will not attempt to look up names for host addresses.
-Q <i>class</i>	Specify the traffic class in the IPv6 header.
-r	Bypass the normal routing tables and send directly to a host on an attached network.
-S <i>sourceAddress</i>	Specify the source address to use when sending.
-s <i>packetSize</i>	Set the number of data bytes to be sent. The default is 56, which translates into 64 ICMP data bytes when combined with the 8 bytes of ICMP header data.
-T	Prefer temporary address as source; sets the IPV6_PREFER_SRC_TMP socket option.
-t	Ping the specified host continuously (this is equivalent to using “-c -1”).
-V <i>routeTable</i>	Set the route table to the one specified by the index number <i>routeTable</i> . The default is 0 (zero). This option is only valid if you have enabled virtual routing.
-w <i>timeout</i>	Set the number of milliseconds between ping requests, default is 1000 ms.

Table 2-2 Ping6 Shell Command Options (cont'd)

Option	Description
<i>hop</i>	0 or more IPv6 addresses of intermediate hosts that should be visited before reaching the final host.
<i>host</i>	The IPv6 address of the final destination node.

When using **ping6** to isolate faults, run it first on the local host, to verify that the local network interface is up and running. Then, ping hosts and gateways further and further away. **ping6** computes round-trip times and packet loss statistics.

2.3 DNS

Domain Name Server (DNS) is a distributed database that applications can use to translate host names to IP addresses and back. DNS uses a client/server architecture. The client is also known as the *resolver*. The server is also called the *name server*.

Technology Overview

The DNS has three major components:

- domain name space and resource records
- name servers
- resolvers

Domain Name Space

The domain name space and resource records specify a tree structured name space and data associated with the names. Conceptually, each node and leaf of the domain name space tree names a set of information, and query operations are attempts to extract specific types of information from a particular set. A query names the domain name of interest and describes the type of resource information that is desired. For example, the Internet uses some of its domain names to identify hosts; queries for address resources return Internet host addresses.

Name servers

Name servers hold information about the domain tree's structure and set information. A name server may cache structure or set information about any part of the domain tree, but in general a particular name server has complete information about a subset of the domain space, and pointers to other name servers that it can use to find information from any other part of the domain tree. Name servers know the parts of the domain tree for which they have complete information; a name server is said to be an authority for these parts of the name space. Authoritative information is organized into units called zones, and these zones can be automatically distributed to the name servers, which provide redundant service for the data in a zone.

Resolvers

Resolvers, or DNS clients, extract information from name servers in response to client requests. Resolvers must be able to access at least one name server and use that name server's information to answer a query either directly, or by pursuing the query using referrals to other name servers. A resolver will typically be a system routine that is directly accessible to your program; hence no protocol is necessary between the resolver and your program.

2.3.1 Component Overview

The Wind River DNS client allows name-to-address lookups as well as address-to-name lookups for systems running IPv4 or dual IPv4/IPv6 stacks. For a list of implemented RFCs and exceptions, see [2.3.2 Conformance to Standards](#), p.13.

The DNS client includes the following features:

- a DNS client (stub resolver) conforming to RFC 1034 and RFC 1035
- one primary and up to three backup name servers
- local caching of resource records
- name-to-address lookups capable of querying for IPv4 addresses as well as IPv6 (RFC 3596) addresses
- address-to-name lookups for IPv4 and IPv6 (RFC 3596) addresses
- the `ipdnsc_getipnodebyname()` and `ipdnsc_getipnodebyaddr()` routines as per RFC 2553

- These routines are reentrant versions of the **gethostbyname()** and **gethostbyaddr()** functions. You can use them to perform DNS lookup of both IPv4 and IPv6 addresses.
- a shell command, **nslookup**, that you can call to perform a name-to-address or address-to-name lookup

2.3.2 Conformance to Standards

The following standards are an incomplete list of the many RFCs describing the DNS. DNS for IPv6 is still not fully standardized and discussions are underway in various IETF Working Groups.

- RFC 1034: *Domain Names OE Concepts and Facilities*
- RFC 1035: *Domain Names OE Implementation and Specification*
- RFC 3596: *DNS Extensions to Support IP Version 6*

Exceptions

Wind River's DNS implementation (IPDNSC) has the following exceptions to the DNS standards:

- Wind River DNS does not allow iterative resolver lookups. The DNS client requires the DNS server to enable recursive lookups.
- Wind River DNS does not allow A6 records.

2.3.3 Configuring and Building DNS

The Wind River Network Stack uses the following DNS resolver configuration components:

DNS Client (**INCLUDE_IPDNSC**)

This component includes the configuration parameters described in [Table 2-3](#).

IPCOM nslookup commands (**INCLUDE_IPNSLOOKUP_CMD**)

This component is described in [Configuring SNTP from the Command Interpreter](#), p.18.

The parameters used to configure DNS are described in [Table 2-3](#).

Table 2-3 Wind River DNS Client Configuration Parameters

Workbench Description, Parameter Name, and sysvar	Default Value and Type
DNS domain name DNSC_DOMAIN_NAME ipdnsc.domainname The domain where the local host is located. If you do not define this parameter, DNS is not aware of the local domain name and DNS users must provide fully qualified domain names in address lookups even for hosts in the same domain as the local host.	"windriver.com" char *
DNS primary name server DNSC_PRIMARY_NAME_SERVER ipdnsc.primaryns The address of the primary name server. If you do not define this parameter, DNS does not use any primary name server.	"" char *
DNS secondary name server DNSC_SECONDARY_NAME_SERVER ipdnsc.secondaryns The address of the secondary name server. If you do not define this parameter, DNS does not use a secondary name server.	"" char *
DNS tertiary name server DNSC_TERTIARY_NAME_SERVER ipdnsc.tertiaryns The address of the tertiary name server. If you do not define this parameter, DNS does not use a tertiary name server.	"" char *
DNS quaternary name server DNSC_QUATERNARY_NAME_SERVER ipdnsc.quaternaryns The address of the quaternary name server. If you do not define this parameter, DNS does not use a quaternary name server.	"" char *

Table 2-3 Wind River DNS Client Configuration Parameters (cont'd)

Workbench Description, Parameter Name, and sysvar	Default Value and Type
DNS server listening port DNSC_SERVER_PORT ipdnsc.port The port that the DNS client uses for DNS queries. If you do not define this parameter, the DNS client uses the default DNS port.	"53" char *
Number of retries for DNS queries DNSC_RETRIES ipdnsc.retries The number of retries on each name server. If you do not define this parameter, DNS uses the default value.	"2" char *
Timeout in seconds when waiting for responses to DNS queries DNSC_TIMEOUT ipdnsc.timeout The number of seconds before a retry if the DNS server fails to answer. If you do not define this parameter, DNS uses the default value.	"10" char *
Zone for IPv4 address to name lookups DNSC_IP4_ZONE ipdnsc.ip4.zone The zone for the DNS client to use for address-to-name lookups of IPv4 addresses.	"in-addr.arpa" char *
Zone for IPv6 address to name lookups DNSC_IP6_ZONE ipdnsc.ip6.zone The zone for the DNS client to use for address-to-name lookups of IPv6 addresses. Use the string ip6.int for RFC 1886 or ip6.arpa for RFC 3152.	"ip6.int" char *

2.3.4 Using nslookup from the Command Interpreter

nslookup—DNS query command

Syntax

```
nslookup [-cf] [-r retries] [-t timeout] [-v version] host [nameServer]
```

Description

nslookup queries an Internet name server for information about a host. The *host* may be a domain name or an Internet v4 or v6 address. The *nameServer* is optional; if you include it (as an Internet address), **nslookup** will use that name server rather than the default.

The command-line options are described in [Table 2-4](#).

Table 2-4 **nslookup Shell Command Options**

Option	Description
-c	Use the DNS resolver’s local cache. The default is to always perform the name server lookup and not to use the cache.
-f	Flush the DNS resolver cache and then exit immediately.
-r <i>retries</i>	Set the number of retries nslookup attempts if the name server fails to answer.
-t <i>timeout</i>	Set how long nslookup waits for an answer from the name server before it retries.
-v <i>version</i>	Set the type of address that is requested in a name-to-address lookup, either 4 (IPv4) or 6 (IPv6). If you do not indicate the version by using the -v flag, nslookup first tries to find an IPv6 address and then, if it cannot find one, it tries to find an IPv4 address.

2.4 SNTP

This section describes the Simple Network Time Protocol (SNTP).

Technology Overview

Network Time Protocol (NTP) synchronizes the clocks of computer systems over packet-switched, variable-latency data networks. The SNTP is a less complex form of NTP that does not store information about previous communication.

You can use SNTP when you do not require a full NTP implementation.

Component Overview

Wind River SNTP is compatible with versions one to four of the SNTP protocol. It handles both IPv4 and IPv6. The SNTP client and server are mutually exclusive components; you may enable only one or the other in a project.

SNTP Client

You can configure the SNTP client to operate in unicast or multicast mode. In unicast mode the client polls SNTP servers for time updates. In multicast mode it listens for SNTP server broadcasts or multicasts.

SNTP Server

The SNTP server is disabled by default. For instructions on enabling it, see [SNTP Server](#), p.21. You can configure the SNTP server to operate in unicast or multicast mode. The unicast mode handles requests from SNTP clients. The multicast mode broadcasts or multicasts SNTP messages periodically, in addition to handling requests from SNTP clients.

SNTP shell command

Wind River SNTP includes a shell command that you can use to update the system time and to do SNTP debugging. See [Configuring SNTP from the Command Interpreter](#), p.18.

Conformance to Standards

Wind River SNTP conforms to RFC 2030: *Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI*.

2.4.1 Configuring and Building SNTP

The Wind River Network Stack includes the following SNTP configuration components:

- IPCOM SNTP commands (**INCLUDE_IPSNTP_CMD**)
- SNTP Client (**INCLUDE_IPSNTPC**)
- SNTP Server (**INCLUDE_IPSNTPS**)
- SNTP common configurations (**INCLUDE_IPSNTP_COMMON**)

Configuring SNTP from the Command Interpreter

sntp—sets the local date and time by using SNTP

Syntax

```
sntp [-bv] [-i interfaceName] [-p port] [-t timeout] {server | mcastAddress}
```

Description

sntp polls an (S)NTP server for the current date and time until it receives a response or the timeout period expires. If it receives a valid response it updates the local date and time. You can also set **sntp** to wait for a (S)NTP server multicast or broadcast.

Mandatory parameters

server

The host name or Internet address of the SNTP server. You must indicate the server unless you use the **-b** or **-i** flags.

mcastAddress

The multicast group that the SNTP client should join if it listens for server multicasts. This parameter is mandatory if you set the **-i** flag.

Options

The options are described in [Table 2-5](#).

Table 2-5 SNTP Shell Command Options

Option	Description
-b	Wait for an (S)NTP server IPv4 broadcast.
-i <i>interfaceName</i>	Wait for (S)NTP server multicasts on the specified interface.
-p <i>port</i>	Set the (S)NTP server's listening port. The default is 123.
-t <i>timeout</i>	Set the maximum time (in seconds) to wait for an answer from the (S)NTP server. A timeout value of negative one means that the program will wait forever. The default is three seconds.
-v	Show the IPSNTP product version.

SNTP Client

The parameters used to configure the SNTP client in Workbench are described in [Table 2-6](#).

Table 2-6 Wind River SNTP Client Configuration Parameters

Workbench Description, Parameter Name, and sysvar	Default Value and Type
Backup server IPv4 address SNTPC_BACKUP_IPV4_ADDR ipsntp.client.backup.addr The IPv4 address of the backup SNTP server.	"10.1.2.40" char *
Backup server IPv6 address SNTPC_BACKUP_IPV6_ADDR ipsntp.client.backup.addr6 The IPv6 address of the backup SNTP server.	"2001::28" char *
Number of retransmissions SNTPC_POLL_COUNT ipsntp.client.poll.count The number of retransmissions the client attempts on each SNTP server.	"3" char *

Table 2-6 Wind River SNTP Client Configuration Parameters (cont'd)

Workbench Description, Parameter Name, and sysvar	Default Value and Type
Primary server IPv4 address SNTPC_PRIMARY_IPV4_ADDR ipsntp.client.primary.addr <p>The IPv4 address of the primary SNTP server.</p>	"10.1.2.90" char *
Primary server IPv6 address SNTPC_PRIMARY_IPV6_ADDR ipsntp.client.primary.addr6 <p>The IPv6 address of the primary SNTP server.</p>	"2001::90" char *
SNTP multicast client mode IPv6 interface SNTPC_MULTICAST_MODE_IPV6_IF ipsntp.client.multi.if6 <p>The SNTP multicast client mode IPv6 interface.</p>	"" char *
SNTP multicast client mode IPv6 multicast group SNTPC_MULTICAST_GROUP_IPV6_ADDR ipsntp.client.multi.addr6 <p>The SNTP multicast client mode IPv6 multicast group.</p>	"ff05::101" char *
SNTP multicast client mode interface SNTPC_MULTICAST_MODE_IF ipsntp.client.multi.if <p>The SNTP multicast client mode interface.</p>	"" char *
SNTP multicast client mode multicast group SNTPC_MULTICAST_GROUP_ADDR ipsntp.client.multi.addr <p>The SNTP multicast client mode multicast group.</p>	"224.0.1.1" char *
SNTP unicast client mode poll interval SNTPC_POLL_INTERVAL ipsntp.client.poll.interval <p>The SNTP client unicast mode poll interval. Set this to zero for multicast mode only.</p>	"1024" char *

Table 2-6 Wind River SNTP Client Configuration Parameters (cont'd)

Workbench Description, Parameter Name, and sysvar	Default Value and Type
Seconds between retransmissions SNTPC_POLL_TIMEOUT ipsntp.client.poll.timeout The number of seconds between retransmissions.	"2" char *
SNTP port SNTP_LISTENING_PORT ipsntp.udp.port The SNTP listening port.	"123" char *

SNTP Server

The SNTP server is disabled by default. The SNTP client and server are mutually exclusive; you may enable only one or the other in a build. The SNTP client (INCLUDE_IPSNTPC) is automatically included if the network stack is built with SNTP.

To enable the SNTP server:

1. Open *installDir/components/ip_net2-6.n/ipsntp/config/ipsntp_config.h*.
2. Comment out the client define:

```
#define IPSNTP_USE_CLIENT
```
3. Uncomment the server define:

```
#define IPSNTP_USE_SERVER
```
4. Re-build the project.

The configuration parameters used to configure the SNTP server in Workbench are described in [Table 2-7](#).

Table 2-7 Wind River SNTP Server Configuration Parameters

Workbench Description, Parameter Name, and sysvar	Default Value and Type
SNTP multicast mode IPv4 destination address SNTPS_IPV4_MULTICAST_ADDR ipsntp.server.mcast.addr The SNTP multicast mode IPv4 destination address.	"10.1.255.255" char *
SNTP multicast mode IPv6 destination address SNTPS_IPV6_MULTICAST_ADDR ipsntp.server.mcast.addr6 The SNTP server multicast mode IPv6 destination address.	"FF05::1" char *
SNTP multicast mode TTL SNTPS_MULTICAST_TTL ipsntp.server.mcast.ttl The SNTP multicast mode time to live.	"1" char *
SNTP multicast mode send interval SNTPS_MULTICAST_INTERVAL ipsntp.server.mcast.interval The SNTP multicast mode send interval in seconds. Set this to zero for unicast mode only.	"3600" char *
SNTP server precision SNTPS_PRECISION ipsntp.server.precision The SNTP server precision. The precision is $2^{\text{SNTPS_PRECISION}}$ seconds.	"-6" char *

Table 2-7 Wind River SNTP Server Configuration Parameters (cont'd)

Workbench Description, Parameter Name, and sysvar	Default Value and Type
SNTP server stratum	"9"
SNTPS_STRATUM	char *
ipsntp.server.stratum	
The SNTP server stratum.	
SNTP port	"123"
SNTP_LISTENING_PORT	char *
ipsntp.udp.port	
The SNTP listening port. SNTP uses this port to send a request to another SNTP server, and also to listen for requests from other clients.	

2.5 FTP

File Transfer Protocol (FTP) is for exchanging files over any TCP/IP network.

Technology Overview

An *FTP server* listens on the network for connection requests from another computer, the *FTP client*, which connects to the FTP server using FTP client software. The FTP client can then manipulate files on the server, for instance uploading, downloading, renaming, and deleting them.

The FTP protocol does not use cryptographically secure communication, so do not use it to transfer confidential data over insecure lines. However, the FTP client and server perform a number of security checks that will prevent some attacks. You can also use FTP over a secure protocol such as IPSec to provide secure file transfer.

Component Overview

The Wind River FTP implementation includes a client and server, and an FTP for backward compatibility. IPv4 and IPv6 are implemented in the same module.

Conformance to Standards

Wind River FTP conforms to the following standards:

- RFC 959: *File Transfer Protocol*
- RFC 1123: *Requirements for Internet Hosts—Application and Support*
- RFC 2428: *FTP Extensions for IPv6 and NAT*
- RFC 2577: *FTP Security Considerations*

Exceptions

Wind River FTP implements only the stream transmission mode of RFC 959, and not the block or compressed transmission modes.

2.5.1 Configuring and Building FTP

The Wind River Network Stack contains the following FTP configuration components:

- FTP Client (**INCLUDE_IPFTPC**)
- FTP Client Backend (**INCLUDE_FTP**)
- FTP Server (**INCLUDE_IPFTPS**)
- FTP6 Client Backend (**INCLUDE_FTP6**)
- IPCOM FTP client commands (**INCLUDE_IPFTP_CMD**)

FTP Client

You can access the Wind River FTP client in two ways—through an API and through the command interpreter (part of the VxWorks kernel shell) using shell commands. This means that an application can select to use only the API, and include customized FTP functionality in proprietary software.

The commands coexist with the API—they actually use the API—which means that both interfaces can be used in the same application.

FTP Client Backend and FTP6 Client Backend

These components are for backward compatibility with a previous release of the Wind River Network Stack, and need not be used for new applications. For details on migrating from a previous release of the network stack see the *Wind River Platforms for VxWorks Migration Guide*.

Table 2-8 Wind River FTP Client Backend Configuration Parameters

Workbench Description and Parameter Name	Default Value and Type
Debug logging facilities in ftpLib FTP_DEBUG_OPTIONS Enable various debugging facilities within ftpLib .	0 int
FTP Transient response maximum retry limit FTP_TRANSIENT_MAX_RETRY_COUNT Maximum number of retries when an FTP_TRANSIENT response is encountered	100 int
FTP timeout FTP_TIMEOUT FTP timeout.	0 long
FTP transient fatal function FTP_TRANSIENT_FATAL Should a transient response be retried or aborted	ftpTransientFatal
Time delay between retries after FTP_TRANSIENT encountered FTP_TRANSIENT_RETRY_INTERVAL The time interval (in clock ticks) between reissuing a command.	0 int
FTP6_REPLYTIMEOUT FTP6_REPLYTIMEOUT The timeout in seconds, while waiting for a reply from the FTP server	10 long

FTP Server

The parameters used to configure the FTP server in Workbench are described in [Table 2-9](#).

Table 2-9 Wind River FTP Server Configuration Parameters

Workbench Description, Parameter Name, and sysvar	Default Value
Authentication attempts before disconnect FTPS_AUTH_ATTEMPTS ipftps.authentications The number of times the server will allow a client to attempt to authenticate itself before the server disconnects.	"3" char *
Authentication callback routine FTPS_AUTH_CALLBACK_HOOK You can use your own routine to authenticate clients. To do this, specify a function pointer for the FTPS_AUTH_CALLBACK_HOOK . The FTP server will call this routine to authenticate clients. The prototype for this routine is as follows: <pre>int myAuthenticateCallback (Ipftps_session * session, char * password);</pre> It should return 0 (zero) if the password is valid for the session, or 1 (one) if you cannot validate the password. If you do not specify an authentication routine, the server will call its own default authentication callback routine that allows read-only access to the user anonymous with no password. If you set a function pointer here, you must also set the FTPS_INSTALL_CALLBACK_HOOK to TRUE in order to install this callback hook.	NULL funcptr
Install ftp server callback routine FTPS_INSTALL_CALLBACK_HOOK Indicates whether the FTP server uses the authentication callback routine that you specified by the configuration parameter FTPS_AUTH_CALLBACK_HOOK to authenticate clients. If this is FALSE , the server instead uses its own authentication routine—one that allows the user anonymous with no password.	FALSE BOOL

Table 2-9 Wind River FTP Server Configuration Parameters (cont'd)

Workbench Description, Parameter Name, and sysvar	Default Value
Data receive timeout FTPS_RECV_TIMEOUT ipftps.receive_timeout The timeout value in seconds when the server is receiving data from the client. If the server does not get any data from the client in this many seconds, the server terminates the connection.	"30" char *
Data send timeout FTPS_SEND_TIMEOUT ipftps.send_timeout The timeout value in seconds when the server is sending data to the client. If the server is unable to transmit any data to the client for this many seconds, the server terminates the connection.	"30" char *
Enable proxy FTP support FTPS_ENABLE_PROXY ipftps.proxy Specifies whether to enable proxy FTP. "1" means enable, "0" means disable.	"0" char *
FTP initial directory FTPS_INITIAL_DIR ipftps.dir The directory on the server that a client starts in when it initiates an FTP session.	IPCOM_FILE_ROOT char *
FTP root directory FTPS_ROOT_DIR ipftps.root The topmost directory that the client is allowed to see.	"/" char *

Table 2-9 Wind River FTP Server Configuration Parameters (cont'd)

Workbench Description, Parameter Name, and sysvar	Default Value
Local port base number FTPS_LOCAL_PORT_BASE ipftps.lportbase <p>Defines the port range the server uses for the data connection. A value of "0" instructs the server to use port 20. The default value of "49151" instructs the server to use ports 49152-65535.</p>	"49151" char *
Max number of simultaneous sessions FTPS_MAX_SESSIONS ipftps.max_sessions <p>The maximum number of simultaneous FTP sessions.</p>	"8" char *
Peer port base number FTPS_PEER_PORT_BASE ipftps.pportbase <p>The base of the port range that the client is allowed to use for PORT and EPRT commands when it establishes the data connection. Set this to "65535" to disable the PORT and EPRT commands entirely (which Wind River recommends for maximum security). The default value, "1023" prevents the server from making any connections to ports 0-1023 on a client.</p>	"1023" char *
Read/write mode FTPS_MODE ipftps.readonly <p>The access mode. Read-only mode disables write access to the file system, which means that only read commands will be accepted. The default value is "0", which means read/write access. Set this to "1" for read-only access.</p>	"0" char *
Reported system type FTPS_SYS_TYPE ipftps.system <p>The operating system name.</p>	"UNIX" char *

Table 2-9 Wind River FTP Server Configuration Parameters (cont'd)

Workbench Description, Parameter Name, and sysvar	Default Value
Server port number FTPS_PORT_NUM ipftps.port_number The port number the FTP server uses for the control connection.	"21" char *
Time to sleep after authentication fail FTPS_SLEEP_TIME ipftps.authsleep How many seconds the server sleeps before it replies to an unsuccessful authentication attempt.	"5" char *
User inactivity timeout FTPS_INACTIVITY_TIMEOUT ipftps.session_timeout The timeout in seconds when the server waits for a command from the client.	"300" char *

2.5.2 Using the FTP Client from the Command Interpreter

ftp—starts and configures the FTP client

Syntax

ftp [-a *family*] [-i] [-l *base*] [-r *base*] [-p *pMode*] [-P *port*] [-v *vMode*] [-x *xMode*] *host*

Description

ftp starts and configures the FTP client.

Options

The command-line options are described in [Table 2-10](#).

Table 2-10 **FTP Client Shell Command Options**

Option	Description
-a <i>family</i>	Set the address family to use when connecting to a server. The default is 0. Possible values are as follows: <ul style="list-style-type: none"> ▪ 0 – IPv4 or IPv6 ▪ 4 – IPv4 only ▪ 6 – IPv6 only
-i	Disable IP address match for control and data. Without this flag, the control and data connection are required to have the same IP address (which disables proxy FTP).
-l <i>base</i>	Set the lower limit for the client data port number. For instance, a value of 1023 means to use a random port in the range 1024-65535. A value of 0 will let the client operating system select a port. The default value is 0.
-r <i>base</i>	Set the lower limit for server data port number. A value of 1023 will only allow server ports in the range 1024-65535. The default value is 1023.
-p <i>pMode</i>	Select passive or active mode. The default value is 3, but reverts to 1 if the server does not use extended mode. <ul style="list-style-type: none"> ▪ 0 – Use only active mode. ▪ 1 – Try passive mode, but revert to active mode if passive mode is not allowed. ▪ 2 – Use only passive mode. ▪ 3 – Lock the server in extended passive mode, which means that the server will require that extended passive mode is used for all future data connections.
-P <i>port</i>	Set the server port number.
-v <i>vMode</i>	Set the verbosity level, which controls the detail of console messages. The default value is 2. <ul style="list-style-type: none"> ▪ 0 – brief mode ▪ 1 – verbose mode ▪ 2 – verbose mode + server output ▪ 3 – verbose mode + client and server output

Table 2-10 FTP Client Shell Command Options (cont'd)

Option	Description
-x <i>xMode</i>	Select the extended mode commands EPRT (extended port) and EPSV (extended passive) for IPv4 connections. The default value is 1. <ul style="list-style-type: none">▪ 0 – Never use extended commands.▪ 1 – First try extended commands, but fall back to normal if those are not allowed.▪ 2 – Always use extended commands.
<i>host</i>	The host name or IP address of the FTP server.

2.6 TFTP

Trivial File Transfer Protocol (TFTP) reads and writes files on a remote server.

Technology Overview

TFTP is a basic form of FTP; it uses a minimal amount of memory, cannot list directory contents, and has no authentication or encryption mechanisms. Unlike FTP, which uses the TCP transport layer protocol (port 21), TFTP uses the UDP (port 69) as its transport protocol.

Component Overview

Wind River TFTP implements the following features:

- a TFTP server and client conforming to RFC 1350
- a TFTP server that listens to both IPv4 and IPv6 connections
- multiple client connections
- both octet and netascii modes
- an API to enable TFTP client operations
- communication over IPv4 or IPv6
- a shell command to transfer files to and from the target

Conformance to Standards

Wind River TFTP conforms to RFC 1350: *The TFTP Protocol (Revision 2)*.

2.6.1 Configuring and Building TFTP

The Wind River Network Stack has the following TFTP configuration components:

- IPCOM TFTP Commands (**INCLUDE_IPTFTP_CMD**)
- TFTP Client (**INCLUDE_IPTFTPC**)
- TFTP Client APIs (**INCLUDE_TFTP_CLIENT**)
- TFTP Server (**INCLUDE_IPTFTPS**)
- TFTP Common Configurations (**INCLUDE_IPTFTP_COMMON**)

Configuring TFTP from the Command Interpreter

tftp—transfer files using TFTP from the shell

Syntax

```
tftp [-a] host {get | put} source [destination]
```

Description

tftp is used to transfer files using the TFTP protocol.

Mandatory parameters

host

The IP address of the host to transfer files with.

get | **put**

Specify **get** to receive a file or **put** to transmit a file.

source

The name of the file to transmit or receive.

Options

-a

Use netascii mode (default is to use binary/octet mode).

destination

The name of the file after it is transmitted or received.

TFTP Client

The TFTP client works over both IPv4 and IPv6. The **INCLUDE_IPTFTPC** component has no configuration parameters. For common TFTP configurations, see [TFTP Common Configurations](#), p.33.

You can invoke the TFTP client through a shell command described in [Configuring TFTP from the Command Interpreter](#), p.32, or from custom programs using the IPTFTPC API. For more information on the functionality associated with this component, see the reference entry for **tftp**.

TFTP Client APIs

Do not use the **INCLUDE_TFTP_CLIENT** API; use the IPTFTPC API. The **INCLUDE_TFTP_CLIENT** component is for backward compatibility with a previous release of the Wind River Network Stack, and to carry forward certain routines not provided in the TFTP client IPTFTPC. For more information on this component, see the reference entry for **tftpLib**.

For details on migrating from a previous release of the network stack see the *Wind River Platforms for VxWorks Migration Guide*.

TFTP Server

The TFTP server works over both IPv4 and IPv6. The **INCLUDE_IPTFTPS** component has no configuration parameters. For common TFTP configuration parameters, see [TFTP Common Configurations](#), p.33.

TFTP Common Configurations

The **INCLUDE_IPTFTP_COMMON** component provides configuration parameters that you can set to determine how TFTP will operate.

The parameters you can set to configure TFTP are described in [Table 2-11](#).

Table 2-11 Wind River TFTP Common Configuration Parameters

Workbench Description, Parameter Name, and sysvar	Default Value
TFTP number of retries TFTPS_RETRIES iptftp.retries The number of times TFTP will resend a message if it does not receive an acknowledgment from the peer. Both the client and the server use this parameter.	"2" char *
TFTP retransmit timeout in seconds TFTPS_RETRANSMIT_TIMEOUT iptftp.timeout The time to wait before until the first retry is sent, in seconds. Further retries use an exponential back-off algorithm. This parameter is used by both the client and server.	"5" char *
TFTP server working directory TFTPS_DIRS iptftp.dir The TFTP server working directory.	IPCOM_FILE_ROOT "tftpDir" char *

2.7 RSH

This section describes the Wind River Network Stack remote shell (RSH).

Technology Overview

RSH allows users to remotely log in to remote systems and execute commands there. The remote system must be running the **rshd** daemon.

Component Overview

The Wind River Network Stack implements the standard RSH client, but does not include an equivalent to the **rshd** daemon. For this reason, remote systems cannot use **rsh** to run commands remotely on a Wind River Network Stack host.

For more information on how to use RSH under the Wind River Network Stack, see the **remLib** reference entries.

2.7.1 Configuring and Building RSH

Add RSH to the Wind River Network Stack by including the Remot Command (**INCLUDE_REMLIB**) configuration component.

The **INCLUDE_REMLIB** component pulls in **remLib**, the library that implements the RSH protocol in VxWorks. Using this RSH implementation, a VxWorks target can execute commands on remote systems that run an **rshd** shell server. The command results return on standard output (**STDOUT**) and standard error (**STDERR**) over socket connections. This library also includes **rcmd_af()**, with which you can execute commands on a remote machine over either IPv4 or IPv6 sockets.

The network stack initialization code calls **remLibInit()** to initialize the RSH client. It passes in the value of the configuration parameter **RSH_STDERR_SETUP_TIMEOUT**:

```
remLibInit (RSH_STDERR_SETUP_TIMEOUT);
```

With this configuration parameter (**Timeout interval for second RSH connection if any**) you can specify how long an **rcmd()** or **rcmd_af()** call waits for a return from its internal call to **select()**. The default value of this configuration parameter is negative one, or **WAIT_FOREVER**, which indicates that there is no timeout.

2.7.2 Enabling Access to an RSH User

An RSH request includes the name of the requesting user. The receiving host can honor or ignore the request based on the user name and the site from which the request originates. How you set up a receiving system to allow access to particular users and sites depends on the specifics of the receiving system's OS and networking software. See that documentation for details.

For UNIX hosts running the BSD **rshd**, an RSH request is honored only if it originated on a known system by a user with local login privileges (though you

can use the **-l** option to configure the BSD **rshd** to omit this verification). The system administrator specifies the list of known systems in either of two locations. The first location, the **/etc/hosts.equiv** file (UNIX) or the **/etc/hosts.allow** file (Linux), maintains a list of all systems from which remote access is allowed for all users that have local accounts. The second location, a **~userName/.rhosts** file, maintains a list of systems from which remote access is allowed for the particular user, *userName*.

Which method you use depends on your security needs. In most environments, adding system names to the **/etc/hosts.equiv** file is considered too dangerous. Thus, for most environments, the preferred method is to add system names to a **~userName/.rhosts** file. The format for this file is one system name per line.

2.8 RPC

This section describes the Wind River Network Stack Remote Procedure Call (RPC) implementation.

Technology Overview

RPC allows a process on one machine to call a procedure that is executed by another process on another machine. It implements a client-server model of task interaction. In this model, client tasks request services of server tasks and then wait for replies. RPC formalizes this model and provides a standard protocol for passing requests and returning replies.

Component Overview

Using RPC, a VxWorks task or host machine process can invoke routines that are executed on other VxWorks or host machines. For more information, see RFC 1831 and the reference entry for **rpcLib**.

The Wind River RPC implementation is a kernel-only implementation. Each task that wants to make an RPC-related call must first call **rpcTaskInit()**.

Conformance to Standards

Wind River RPC conforms to RFC 1831: *RPC: Remote Procedure Call Protocol Specification Version 2*.

2.8.1 Configuring and Building RPC

The Wind River Network Stack includes the following RPC configuration components:

RPC

The **INCLUDE_RPC** component pulls in **rpcLib** and other modules that implement RPC. For information on the API associated with the modules in this component, see the **rpcLib** reference entry.

XDR

The **INCLUDE_XDR** component pulls in modules that implement generic XDR (External Data Representation) routines as described in RFC 1014. There is no API or configuration parameters associated with this component.

XDR boolean support

The **INCLUDE_XDR_BOOL_T** component pulls in modules that supply the XDR routine for **bool_ts**. There is no API or configuration parameters associated with this component.

2.9 rlogin

This section describes the Wind River Network Stack **rlogin** implementation. **rlogin** allows remote shell access to and from a target. The Wind River implementation of **rlogin** works over both IPv4 and IPv6.

2.9.1 Configuring and Building rlogin

The **RLOGIN** (**INCLUDE_RLOGIN**) component of the Wind River Network Stack enables **rlogin**. To password-protect the rlogin shell, you must also include **TELNET/FTP password protection** (**INCLUDE_SECURITY**).



CAUTION: If you include the **INCLUDE_SECURITY** component for rlogin authentication, change the default user name and password to prevent unauthorized access from a source that is familiar with the default settings (see [rlogin password protection](#), p.38).

RLOGIN

The `INCLUDE_RLOGIN` component pulls in `rlogLib`, a library that implements `rlogin`. This component requires the `INCLUDE_NET_HOST_SETUP` component.

Using the client, you can log in to a host system from a VxWorks terminal. Using the server, you can log in to VxWorks from a host system. For more information on the Wind River Network Stack implementation of `rlogin()`, see the reference entry for `rlogLib`.

rlogin password protection

The `INCLUDE_SECURITY` component provides rlogin password protection. This component depends on the configuration parameters `LOGIN_USER_NAME` and `LOGIN_PASSWORD` to supply a user name and a password for that user name. These parameters are described in [Table 2-13](#).

Table 2-12 Wind River rlogin Password Configuration Parameters

Workbench Description and Parameter Name	Default Value
rlogin/telnet encrypted password <code>LOGIN_PASSWORD</code> The rlogin password.	"RcQbRbzRyc"
rlogin/telnet user name <code>LOGIN_USER_NAME</code> The rlogin user name.	"target"

Connecting to Host Systems

When a VxWorks client system connects to a Windows host system, the ability of VxWorks to remotely log in depends on the version of Windows and the networking software on the Windows host. See that documentation for details.

For a VxWorks client to connect to a UNIX host system, the UNIX host must grant access permission to the VxWorks client by entering its system name either in the `.rhosts` file (in the client's home directory) or in the `/etc/hosts.equiv` file. For more information, see [2.7.2 Enabling Access to an RSH User](#), p.35.

2.10 Telnet

Telnet provides command-line login sessions between hosts on a network. The Wind River implementation of telnet works over both IPv4 and IPv6.

Conformance to Standards

Wind River telnet conforms to RFC 854: *Telnet Protocol Specification*.

2.10.1 Configuring and Building Telnet

The Wind River Network Stack has the following telnet configuration components:

- TELNET client (**INCLUDE_TELNET_CLIENT**)
- TELNET/FTP password protection (**INCLUDE_SECURITY**)
- Telnet Server (**INCLUDE_IPTELNETS**)



CAUTION: If you include the **INCLUDE_SECURITY** component for telnet authentication, change the configuration parameters **LOGIN_USER_NAME** and **LOGIN_PASSWORD** to prevent unauthorized access from a source that is familiar with the default settings.

TELNET client

The **INCLUDE_TELNET_CLIENT** component pulls in the **telnetcLib** module and the VxWorks **telnet** client, a lightweight implementation of the client side of the telnet protocol (RFC 854). This client works over both IPv4 and IPv6, provides all the essential features of NVT (Network Virtual Terminal), and can function as an interface between terminal and terminal-oriented processes. For details, see the **telnetcLib** reference entry.



NOTE: By default, the client sends an **ECHO** at start up. It works in default mode only when it receives **DO** and **DONOT** from a remote **telnet** server.

TELNET/FTP password protection

The **INCLUDE_SECURITY** component provides telnet password protection. This component depends on the configuration parameters **LOGIN_USER_NAME** and **LOGIN_PASSWORD** to supply a user name and a password for that user name.

The parameters that you use to configure the user name and password are described in [Table 2-13](#).

Table 2-13 **Wind River Telnet Password Configuration Parameters**

Workbench Description and Parameter Name	Default Value
rlogin/telnet encrypted password LOGIN_PASSWORD The telnet password.	"RcQbRbzRyc"
rlogin/telnet user name LOGIN_USER_NAME The telnet login user name.	"target"

Telnet Server

The telnet server enables other machines or users to remotely log in to the target and access the target’s shell. This allows those remote users to execute shell commands and programs whose output and input are displayed on the remote terminal. The telnet server connects by default to the VxWorks shell.

The telnet server implements the following features:

- allows multiple connections
- listens to connections from both IPv4 and IPv6 sockets if these are available in the TCP/IP stack
- authenticates users
- allows users to connect to an underlying shell
- optionally echoes user-typed characters

The parameters you can use to configure the telnet server are described in [Table 2-14](#).

Table 2-14 Wind River Telnet Server Configuration Parameters

Workbench Description and Parameter Name	Default Value and Type
IPCOM telnet port IPCOM_TELNET_PORT The telnet server port in host endian.	"23" char *
Telnet authentication IPCOM_TELNET_AUTH_ENABLED Specifies whether to use authentication for telnet connections. Set this to 1 to require telnet users to be authenticated. This parameter is only meaningful if the stack is built with INCLUDE_IPCOM_USE_AUTH .	"0" char *

2.11 Creating a netDrv Device for RSH or FTP

The Wind River Network Stack provides an implementation of an RSH client. The Wind River Network Stack also provides an implementation of both an FTP client and an FTP server.

A VxWorks application can use RSH to run commands on a remote system and receive the command results on standard output and standard error over socket connections. To execute commands remotely, RSH requires that the remote system runs the server side of RSH and that the remote system grant access privileges to the user specified in the RSH request. On a UNIX system, the RSH server is implemented using the **rshd** shell daemon, and access privileges are controlled by a **.rhosts** file. On a Wind River Network Stack host, there is no equivalent to **rshd**. Thus, remote systems cannot use RSH to run commands on a Wind River Network Stack host.

Using netDrv Drivers

You can use RSH and FTP directly, but you can also use them indirectly to download files through the mediation of the **netDrv** driver. Using **netDrv** in this way is especially convenient when a target needs to download a run-time image at boot time.

The **netDrv** driver facilitates access to remote files over the network using RSH and FTP. If you create a network device with **netDevCreate()**, you can access files on a remote UNIX machine as if they were local.

When you open a remote file through a **netDrv** driver, this copies the entire file over the network into a local buffer. When you create a remote file, this opens an empty local buffer. Any reads, writes, or **ioctl()** calls that you make apply only to the local copy of the file. If you modify the file, when you close it the local copy will be sent back over the network to overwrite the copy on the remote UNIX machine (unless you opened the file in read-only mode).

Setting the User ID for Remote File Access with RSH or FTP

Since the **netDrv** driver uses RSH or FTP to provide access to the remote file system, in order to use a **netDrv** driver to access remote files you must set your user name and password to what the remote system expects for RSH or FTP clients. From VxWorks, you can specify the user name and password for remote requests by calling **iam()**:

```
iam ("userName", "password");
```

The first argument to **iam()** is the user name that identifies you when you access remote systems. The second argument is the FTP password. This is ignored if RSH is being used, and can be specified as **NULL** or 0 (zero).

For example, the following command tells VxWorks that all accesses to remote systems with RSH or FTP are through user *darger*, and if FTP is used, the password is *unreal*:

```
-> iam "darger", "unreal"
```



NOTE: When a VxWorks boot program downloads a run-time image from a remote network source using a **netDrv** instance, it relies upon either the FTP or RSH protocols. The boot program relies upon the values that you specified for the user name and password in the boot line.

Setting File Permissions on the Remote System

For a VxWorks system to access a particular file on a host, you must set permissions on the host system appropriately. The user name as seen from the host must have permission to read that file (and write it, if necessary). That user name must also have permission to access all directories in the path to the file. The easiest way to check this is to log in to the host with the user name VxWorks uses, and try to read or write the file in question. If you cannot do this, neither can the VxWorks system.

Creating a netDrv Instance

Although you can use **netDrv** at boot time to download a run-time image, **netDrv** is not limited to boot time or run-time images. It is a generic I/O device that you can use to access files on a remote networked system. To include **netDrv**, use the **INCLUDE_NET_DRV** configuration component.

To use **netDrv**, you must create a **netDrv** instance for each system on which you want to access files. You can then use this device in standard VxWorks I/O device calls such as **open()**, **read()**, **write()**, and **close()**. To create a **netDrv** device, call **netDevCreate()**:

```
netDevCreate ( "deviceName", "host", protocol );
```

Its arguments are:

deviceName

The name of the device to be created. Typically, you compose the device name using the host name followed by a colon.

host

The Internet address of the host in dot notation, or the name of the remote system as specified in a previous call to **hostAdd()**.

protocol

The file transfer protocol: 0 for RSH or 1 for FTP.

For example, the following call creates a new I/O device on VxWorks called **mars:**, which accesses files on the host system **mars** using RSH:

```
-> netDevCreate "mars:", "mars", 0
```

After a network device is created, files on that host are accessible by appending the host pathname to the device name. For example, the filename **mars:/usr/darger/myfile** refers to the file **/usr/darger/myfile** on the **mars** system. You can read or write to this file as if it were a local file. For example, the following shell command opens that file for I/O access:

```
-> fd = open ("mars:/usr/darger/myfile", 2)
```

Using netDrv to Download Run-Time Images

The VxWorks network startup routine, **usrNetInit()** in **usrNetwork.c**, in a VxWorks boot program automatically creates a **netDrv** instance for the host name that you specified in the VxWorks boot parameters. The boot program then uses this device to download an image from that host.

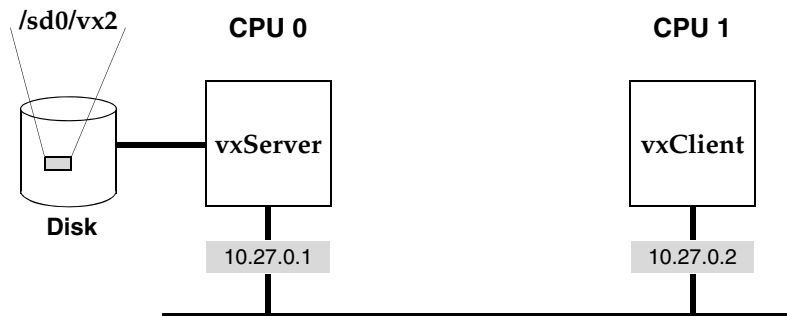
Whether the **netDrv** instance uses FTP or RSH to download the image depends on whether the boot parameters include an FTP password. If an FTP password is

present, **netDrv** uses FTP; otherwise, **netDrv** uses RSH. When you create a boot program that must download an image using an RSH or an FTP client, you must include the RSH or FTP client components in your build.

Consider the system shown in [Figure 2-1](#). CPU 1 can use FTP to download its run-time image from the storage device accessible through CPU 0. Note that **vxClient** must include a non-empty **ftp password** field in its boot parameter. By including an FTP password, **vxClient** tells **netDrv** to use FTP.

Whether the FTP server on CPU 0 checks the validity of the password depends on whether that FTP server has been configured with security turned off (the default) or on.

Figure 2-1 **FTP Boot Example**



3

Wind River DHCP and DHCPv6: Overview

3.1 Introduction

The Dynamic Host Configuration Protocol (DHCP) is an Internet protocol that automates the configuration of computers that use TCP/IP. You can use DHCP to automatically assign IP addresses, deliver TCP/IP stack configuration parameters such as the subnet mask and default router, and provide other configuration information, such as the addresses for printer, time, and news servers. DHCP is specified in RFC 2131: *Dynamic Host Configuration Protocol* and RFC 2132: *DHCP Options and BOOTP Vendor Extensions*.

The Wind River Network Stack includes a server, client, and relay agent for both DHCP and DHCPv6. The chapters that follow this overview describe these implementations.

DHCP extends an earlier protocol, Bootstrap Protocol (BOOTP, see RFCs 951 and 1542), and you can configure most DHCP servers, including the Wind River DHCP server, to allow requests from BOOTP clients.

When a DHCP server assigns an IP address to a client, it gives the client a *lease* on that address for a specific length of time. Clients can renew their leases before they expire. When a lease expires, the server can reassign the IP address to a different computer. This allows the server to maintain a pool of IP addresses that it reuses as computers join and leave the network.

Although the primary purpose of DHCP is dynamic configuration, it can also assign a client a static, permanent IP address and configuration. In such a case, the client is given a lease of infinite duration.

3.1.1 Architectural Overview: Client, Server, and Relay Agent

DHCP uses a client-server model. DHCP clients obtain configuration information from a DHCP server. DHCP servers maintain databases with lease and configuration information and provide this information to DHCP clients.

A DHCP client obtains configuration information by broadcasting a message to locate one or more DHCP servers. Routers are generally configured to block broadcast messages, which would prevent a client's broadcast message from reaching a DHCP server that is not on the same subnet as the client. DHCP overcomes this problem through the use of a *relay agent*. The relay agent relays messages from the client's subnet to servers on other subnets. Typically the relay agent resides on a router, but this is not a requirement.

A DHCP client may receive lease offers from multiple servers, at which point it may choose the offer that best meets its requirements.

3.1.2 DHCPv6

DHCPv6 (RFC 3315: *Dynamic Host Configuration Protocol for IPv6*) is DHCP for IPv6. In its functionality and architecture DHCPv6 is similar to DHCP for IPv4, but there are important differences. Many of these reflect the change in addressing from IPv4 to IPv6, but there are also changes in other areas. For example, DHCPv6 introduces changes in messaging and message formats and you configure it with a different set of configuration options.

3.1.3 Build Configuration Parameters and sysvars

The DHCP-related configuration components provides several configuration parameters, which are listed in tables in the following chapters. For each parameter, the table gives the Workbench name, macro name, and sysvar.

At run time, you can set most of these parameters, and many additional ones, through shell commands (see [4.3.2 Configuring the Server with Shell Commands](#), p.62) or by implementing a routine that reads a configuration file at startup (see [Example 4-3](#)).

Note the following characteristics of parameters and parameter values:

- Those parameters that are strings (type = char *) must be entered as quoted strings, including the quotation marks.

- Many parameters allow a semicolon-separated list of entries in the following format:

```
"interfaceName=value; interfaceName=value; interfaceName=value"
```

The following is an example (see the table entry for **Interface Status List**, p. 123):

```
"eth0=automatic; eth1=enable; vlan21=enable"
```

- If a static configuration parameter accepts a list of entries, you can use the corresponding sysvar shell command multiple times to enter parameter values.

For example, the following sequence of shell commands accomplishes the same thing as the **Interface Status List** parameter setting example in the preceding bullet item:

```
-> sysvar set ipdhcpc6.if.enum.eth0 automatic  
-> sysvar set ipdhcpc6.if.enum.eth1 enable  
-> sysvar set ipdhcpc6.if.enum.vlan21 enable
```

- There is no default value for a parameter that allows a list of entries, but in many cases, there is a default value for the *value* side of an *interfaceName=value* pair. In such cases, this is the value shown in the "Default Value" column of the table.

For example, the **Interface Information Refresh Min Status List** (DHPC6_IF_INFO_REFRESH_MIN_LIST) parameter allows you to list individual interfaces and time intervals for refreshing stateless information. The default time interval is 600 seconds. Therefore, the "Default Value" column shows "600", and you do not need to list interfaces that use the default interval.

4

Wind River DHCP: Server

- 4.1 Introduction 49
- 4.2 Including the DHCP Server in a Build 52
- 4.3 Setting Up Addresses, Options, Subnets and Hosts 59
- 4.4 Implementing Hook Routines for Initialization and Shutdown 66
- 4.5 Setting Options in Shell Commands and API Routines 72

4.1 Introduction

This chapter describes the Wind River DHCP server implementation. For a general overview of DHCP, see the preceding chapter, [3. Wind River DHCP and DHCPv6: Overview](#).

Conformance to Standards

The Wind River DHCP server implements the server portions of RFC 2131: *Dynamic Host Configuration Protocol*, and it also implements certain options and vendor extensions in RFC 2132: *DHCP Options and BOOTP Vendor Extensions* and RFC 2242: *NetWare/IP Domain Name and Information*.

RFC 2131, Dynamic Host Configuration Protocol

The server implements all server-related features of RFC 2131.

RFC 2132, DHCP Options and BOOTP Vendor Extensions

The server implements all options and vendor extensions in RFC 2132, with the exception of the following options:

- 60 – Vendor class identifier
- 66 – TFTP server
- 67 – Bootfile name

There is a Wind River-specific option for setting a bootfile name; see the entry for **boot-file** in [Table 4-4](#).

RFC 2242, NetWare/IP Domain Name and Information

RFC 2242 adds two option codes for the NetWare/IP product: Option code 62 (NetWare/IP Domain Name) and option code 63 (Netware/IP Information). The Wind River DHCP server implements option code 62, but does not implement option code 63.

4.1.1 Server Overview

The Wind River DHCP server provides IP addresses and option settings to DHCP clients. The option settings are of two types:

- DHCP options, as defined in RFC 2132 and RFC 2242 (see [4.5 Setting Options in Shell Commands and API Routines](#), p.72, [Table 4-3](#))
- Wind River-specific configuration options (see [4.5 Setting Options in Shell Commands and API Routines](#), p.72, [Table 4-4](#))

You can configure the information the DHCP server provides to clients in three ways:

- **Set global defaults**

You can set defaults for all options. You can also specify a default set of IP addresses if your DHCP server provides addresses only to clients on a single subnet.

- **Assign IP addresses and options on a subnet-by-subnet basis, dynamically**

For each subnet your DHCP server covers, you can specify options and a set of addresses for clients on that subnet. The option settings you specify for a subnet override any global default settings for the same options.

- **Assign IP addresses and options to individual hosts, statically**

You can also assign a static IP address and an individualized set of options to individual clients. The option settings you specify for an individual host override any global default settings or subnet settings for the same options.

You can configure information about addresses, options, subnets, and hosts, both statically at build time and dynamically at run time, see [4.3 Setting Up Addresses, Options, Subnets and Hosts](#), p.59.

4.1.2 Server Components

The Wind River DHCP server includes a configuration database, a lease database and a single process, the DHCP server daemon.

The DHCP Daemon

The DHCP daemon handles client requests in the following way:

1. It checks each packet received to ensure that it is a DHCP or BOOTP packet.
2. It looks for a subnet in its configuration database that matches the client's subnet. If the request came directly from the client, the server interprets the subnet of the interface on which the request was received as the client's subnet. If the server receives the request through a relay agent, the server interprets the relay agent's subnet as the client's subnet.
3. It scans its lease database for a lease matching the client's identification key (which is either the client's link-layer address or a client identifier). If the server finds such a lease, it responds to the client's request based on the lease. If the server does not find a lease, it offers the client a new lease based on the client's subnet and the client's request.

The Configuration Database

The configuration database stores the information you configure for global default values, subnets, individual hosts, and options.

The Lease Database

The lease database contains all leases currently in effect and all leases previously used by clients that are now either expired or released. The server uses this database to keep track of the state of clients with active leases and to give clients whose leases are no longer in effect the same addresses and options they had in the past.

You can save and restore this lease database so that the server does not have to rebuild it from scratch each time it starts up. The **ipdhcps_lease_db_dump()** routine dumps the database to a buffer (you need to implement additional operations to write this database buffer to a file). The **ipdhcps_lease_db_restore()** routine restores the database from a buffer (you need to implement whatever operations are necessary to get the database data into the buffer in the first place).

To restore the database at startup, implement the **ipdhcps_start_hook()** routine to call **ipdhcps_lease_db_restore()**. For an example, see [Example 4-2 Restoring the Lease Database at Startup](#), p.67. The server calls **ipdhcps_start_hook()** when it starts.

To dump the database when the server shuts down, implement the **ipdhcps_stop_hook()** routine to call **ipdhcps_lease_db_dump()**. For an example, see [Example 4-4 Dumping the Lease Database at Shutdown](#), p.70. The server calls **ipdhcps_stop_hook()** when you shut the server down.

4.2 Including the DHCP Server in a Build

To include the DHCP server in a VxWorks build, include the **DHCP Server** (**INCLUDE_IPDHCP**) build component.

DHCP Server Configuration Parameters

The **DHCP Server** (**INCLUDE_IPDHCP**) build component has a number of configuration parameters. [Table 4-1](#) lists and describes each parameter. See [3.1.3 Build Configuration Parameters and sysvars](#), p.46 for more information on setting these parameters.

Table 4-1 DHCP Server Build Parameters

Workbench Description, Parameter Name, and sysvar	Default Value & Data Type
UDP port used by the DHCP server DHCPS_SERVER_PORT ipdhcps.server_port <p>The port on the DHCP server to which a client sends DHCP messages.</p>	"67" char *
UDP port used by the dhcp/bootp clients DHCPS_CLIENT_PORT ipdhcps.client_port <p>The port on a DHCP/BOOTP client to which the DHCP server sends DHCP messages.</p>	"68" char *
Default lease time DHCPS_DEFAULT_LEASE_TIME ipdhcps.default_lease_time <p>The default length of time, in seconds, that a client can hold a dynamic lease before it expires, or "forever" to indicate a permanent lease. You can also configure this at run time.</p>	"864000" (10 days) char *
Maximum lease time DHCPS_MAX_LEASE_TIME ipdhcps.max_lease_time <p>The maximum length of time, in seconds, that a client can hold a dynamic lease before it expires, or "forever" to indicate a permanent lease. You can also configure this at run time.</p>	"8640000" (100 days) char *
Minimum lease time DHCPS_MIN_LEASE_TIME ipdhcps.min_lease_time <p>The minimum length of time, in seconds, that a client can hold a dynamic lease before it expires. You can also configure this at run time.</p>	"60" char *

Table 4-1 **DHCP Server Build Parameters** (cont'd)

Workbench Description, Parameter Name, and sysvar	Default Value & Data Type
Lease renewal time DHCPS_RENEWAL_TIME ipdhcps.renewal_time <p>Either "disabled" or the length of time, in seconds, after which a client must request its lease to be extended, if it wants to keep its IP address. If set to "disabled", the server does not send a renewal time to clients. You can also configure this at run time.</p>	"disabled" char *
Lease rebinding time DHCPS_REBINDING_TIME ipdhcps.rebinding_time <p>Either "disabled" or the length of time, in seconds, after which a client must attempt to rebind a lease if it has failed to renew it. If set to "disabled", the server does not send a rebinding time to clients. You can also configure this at run time.</p>	"disabled" char *
Abandoned state max time DHCPS_ABANDONED_STATE_MAX_TIME ipdhcps.in_abandoned_state_max_time <p>The length of time, in seconds, that must pass before an abandoned lease can be released and is free for reuse, or "forever" to indicate that the lease must not be released. A lease is considered to be abandoned when a client has explicitly declined it because the address bound to the offered lease is already in use by another host.</p>	"8640000" (100 days) char *
Expired state max time DHCPS_EXPIRED_STATE_MAX_TIME ipdhcps.in_expired_state_max_time <p>Either "forever" or the length of time, in seconds, that must pass before an expired lease can be assigned to another client. When a lease expires, the server keeps it in the lease database in case the client it belonged to wants to claim it again.</p>	"forever" char *

Table 4-1 DHCP Server Build Parameters (cont'd)

Workbench Description, Parameter Name, and sysvar	Default Value & Data Type
Released state max time DHCPS_RELEASED_STATE_MAX_TIME ipdhcps.in_release_state_max_time <p>Either "forever" or the length of time, in seconds, that must pass before a released lease can be assigned to another client. The server puts the lease in the released state when the client holding it tells the server to release it. The server holds it in the lease database in case the client wants to claim it again.</p>	"forever" char *
Offered state max time DHCPS_OFFERED_STATE_MAX_TIME ipdhcps.in_offered_state_max_time <p>The length of time, in seconds, that must pass before the server assigns an a lease that is in the "offered" state to another client, or "forever" to indicate that the lease may not be assigned to another client. A lease is in an offered state when the server has offered it to a client but that client has not yet bound to it.</p>	"10" char *
Lease for bootp client max time DHCPS_LEASE_BOOTPC_MAX_TIME ipdhcps.in_bootp_state_max_time <p>Either "forever" or the length of time, in seconds, that must pass before the server can assign a lease held by a BOOTP client to another client.</p>	"forever" char *

Table 4-1 **DHCP Server Build Parameters** (cont'd)

Workbench Description, Parameter Name, and sysvar	Default Value & Data Type
Allow decline DHCP_ALLOWED_DECLINE ipdhcps.allow_decline Either "true" or "false". If "true", the server accepts DHCP Decline messages. If "false", the server silently ignores such messages. Clients send messages of type DHCP Decline when they want to notify the server that an IPv4 address conflict has occurred, and the address offered by the server is already in use by another host. The default server response is to consider the lease containing the address to be abandoned. However, this makes the server vulnerable to a malfunctioning client or a denial of service attack that could empty the server's address pools. For this reason, you can set the server to ignore client declines.	"true" char *
Packet size DHCP_PKT_SIZE ipdhcps.packet_size The size, in bytes, of packets sent by the DHCP server. The value must be at least 576 bytes. If you set this to a higher value, when a client contacts the server, the server informs the client of the packet size.	"576" char *
Allow bootp DHCP_ALLOWED_BOOTP ipdhcps.allow_bootp Either "true" or "false". If "true", the server accepts requests from BOOTP clients. You can also configure this at run time.	"true" char *

Table 4-1 DHCP Server Build Parameters (cont'd)

Workbench Description, Parameter Name, and sysvar	Default Value & Data Type
Allow dynamic bootp DHCPS_ALLOW_DYNAMIC_BOOTP ipdhcps.allow_dynamic_bootp <p>Either "true" or "false". If "true", the server may assign dynamic leases to BOOTP clients. If set to "false", the server assigns only static leases to BOOTP clients. Because BOOTP clients are not able to release their leases, Wind River recommends that you always set Allow dynamic bootp to "false". You can also configure this at run time.</p>	"false" char *
Check address DHCPS_DO_ICMP_ADDRESS_CHECK ipdhcps.do_icmp_address_check <p>Either "true" or "false". If "true", the server sends an ICMP ping to check an address before offering it to a client.</p>	"false" char *
Authorized dhcp relay agent DHCPS_AUTHORIZED_AGENTS ipdhcps.authorized_agents <p>Either "any" or a space-delimited list of IP addresses of DHCP relay agents that are authorized to access the server.</p>	"any" char *
DHCP server network pre-configuration DHCPS_NETCONF_SYSVAR sysvar: None. <p>Either "true" or "false". If "true", you can statically configure the server by entering configuration values in the ipdhcps_netconf_sysvar array in the ipdhcps_config.c file. For information on configuring the server in this way, see 4.3.1 Configuring the Server with the ipdhcps_netconf_sysvar Array, p.59.</p>	NULL void *

Table 4-1 **DHCP Server Build Parameters** (cont'd)

Workbench Description, Parameter Name, and sysvar	Default Value & Data Type
Install dhcp server callback routines DHCPS_INSTALL_CALLBACK_HOOK sysvar: None. If set to TRUE , you can include your versions of the ipdhcps_start_hook() and ipdhcps_stop_hook() routines in the build.	FALSE BOOL
DHCP server startup callback routine DHCPS_START_CALLBACK_HOOK sysvar: None. A function pointer to an implementation of the ipdhcps_start_hook() routine. The server calls this hook routine at startup. You can use this routine to perform operations such as restoring the lease database from a file and reading parameter settings from a configuration file (see 4.4.1 The ipdhcps_start_hook() Routine , p.66). If you set this parameter, you must also set the Install dhcp server callback routines parameter to TRUE .	NULL funcptr
DHCP server termination callback routine DHCPS_STOP_CALLBACK_HOOK sysvar: None. A function pointer to an implementation of the ipdhcps_stop_hook() routine. The server calls this hook routine just before the DHCP daemon exits. You can use this routine to dump the lease database to a file and perform other cleanup operations (see 4.4.2 The ipdhcps_stop_hook() Routine , p.70). If you set this parameter, you must also set the Install dhcp server callback routines parameter to TRUE .	NULL funcptr

4.3 Setting Up Addresses, Options, Subnets and Hosts

You can configure the DHCP server's initial state—IP addresses, options, subnets, hosts, and default settings—either statically or dynamically.

To statically configure DHCP, enter the configuration settings in the **ipdhcps_netconf_sysvar** array, see [4.3.1 Configuring the Server with the ipdhcps_netconf_sysvar Array](#), p.59.

To dynamically configure DHCP, either issue shell commands (see [4.3.2 Configuring the Server with Shell Commands](#), p.62) or implement the **ipdhcps_start_hook()** routine to read a configuration file at startup (see [4.4.1 The ipdhcps_start_hook\(\) Routine](#), p.66).

4.3.1 Configuring the Server with the ipdhcps_netconf_sysvar Array

You can statically configure the server by entering commands and parameter values in the **ipdhcps_netconf_sysvar** array. To do this you must enable this array by setting the **DHCP server network pre-configuration** (DHCP_SERVER_NETWORK_PRECONFIG) build parameter to "true" (see [DHCP Server Configuration Parameters](#), p.52, [Table 4-1](#)).

You can find the **ipdhcps_netconf_sysvar** array in the **ipdhcps_config.c** file. For a list of valid commands, see [Commands Available in the ipdhcps_netconf_sysvar Array](#), p.60. The path to **ipdhcps_config.c** is:

```
installDir/components/ip_net2-6.n/osconfig/vxworks/src/ipnet/ipdhcps_config.c
```

With the exception of the last entry in the array, entries have the following form:

```
{"ipdhcps.netconf.index", "command arguments"}
```

where

index is an index into the **ipdhcps_netconf_sysvar** array, and

command is a two-word expression, such as the command **subnet add**, which is used to add a subnet to the configuration database.

The last entry in the array must always be:

```
{IP_NULL, IP_NULL}
```

The following example adds subnet 10.1.0.0 with subnet mask 255.255.0.0 to the configuration database in the first element in the **ipdhcps_netconf_sysvar** array:

```
IP_STATIC IP_CONST Ipcom_sysvar ipdhcps_netconf_sysvar[] =
{
    {"ipdhcps.netconf.00", "subnet add 10.1.0.0 255.255.0.0"},
    ...
    {IP_NULL, IP_NULL}
}
```

Commands Available in the ipdhcps_netconf_sysvar Array

You can use the following commands in the DHCP static configuration array:

`subnet add netAddress mask`

Add a subnet with the specified network address and mask to the configuration database. The following example adds the 10.1.x.x subnet:

```
{"ipdhcps.netconf.00", "subnet add 10.1.0.0 255.255.0.0"}
```

`pool add subnet firstAddress lastAddress`

Add an address pool to a subnet, for example:

```
{"ipdhcps.netconf.01", "pool add 10.1.0.0 10.1.0.5 10.1.0.99"}
```

`host add hostIPAddress [-h linkLayerAddress | -i clientID]`

Add the specified host to the configuration database. To specify the host, use the host's IP address and, optionally, a link-layer address or a client identifier (see [dhcps host](#), p.63). The following example adds a host with the client identifier of "LabSys1" with the IP address 10.0.0.142:

```
{"ipdhcps.netconf.13", "host add 10.0.0.142 -i \"LabSys1\""}
```

`option add address optionCode value`

Add a DHCP option, as listed in RFC 2132, to the server's default set of options or to the options associated with a specified host address or subnet address. To add an option to the default set, use **default** as the address.

For a table of the options available on the Wind River DHCP server, see [4.5.1 Using Standard DHCP Options in Shell Commands and APIs](#), p.72, Table 4-3. The following DHCP options are not available:

- Vendor class identifier (option code 60)
- TFTP server (option code 66)
- Bootfile name (option code 67; there is a Wind-River specific option for setting a bootfile name; see the entry for **boot-file** in Table 4-4)

The following example adds two routers to subnet 192.168.2.x. The DHCP option code to add routers is 3.

```
{"ipdhcps.netconf.09", "option add 192.168.2.0 3 192.168.2.1  
192.168.2.2"}
```

`config set address optionCode value`

The **config set** command applies to options that are specific to Wind River's DHCP implementation. For information on these options, see [4.5.2 Using Wind River-Specific Options in Shell Commands and APIs](#), p.80, Table 4-4.

This command adds a **dhcps config** option to the default set of options or to the options associated with a specified host address or subnet address. To add an option to the default set, use **default** as the address.

The following example specifies a boot file for host 10.0.0.141:

```
{ "ipdhcps.netconf.18", "config set 10.0.0.141 boot-file
/boot/hostZ.boot" }
```

Example 4-1 Sample ipdhcps_netconf_sysvar Array

The following example shows sample entries in an **ipdhcps_netconf_sysvar** array:

```
IP_STATIC IP_CONST Ipcom_sysvar ipdhcps_netconf_sysvar[] =
{
    /* Add subnet 192.168.2.0 with netmask 255.255.255.0 */
    {"ipdhcps.netconf.01", "subnet add 192.168.2.0 255.255.255.0"},

    /* Add an address pool to the subnet */
    {"ipdhcps.netconf.02", "pool add 192.168.2.0 192.168.2.10 192.168.2.50"},

    /* Add a domain name (dhcp option 15) to the subnet */
    {"ipdhcps.netconf.03", "option add 192.168.2.0 15 blue.net"},

    /* Add two routers (dhcp option 3) to the subnet */
    {"ipdhcps.netconf.04",
     "option add 192.168.2.0 5 192.168.2.1 192.168.2.2"},

    /*
     * Add a static route (dhcp option 33) to the subnet
     * (10.1.0.0->192.168.2.2)
     */
    {"ipdhcps.netconf.05", "option add 192.168.2.0 33 10.1.0.0 192.168.2.2"},

    /* Set max lease time to 120 seconds on the subnet */
    {"ipdhcps.netconf.06", "config set 192.168.2.0 max-lease-time 120"},

    /* Add a host with the client identifier 'HostOne' */
    {"ipdhcps.netconf.07", "host add 10.0.0.142 -i \"HostOne\""},

    /* Add a host name (dhcp option 12) for HostOne */
    {"ipdhcps.netconf.08", "option add 10.0.0.142 12 blue-host"},

    /* Add a router (dhcp option 3) to HostOne */
    {"ipdhcps.netconf.09", "option add 10.0.0.142 3 10.1.0.1"},

    /* Add a host with a standard Ethernet MAC address */
    {"ipdhcps.netconf.10", "host add 10.0.0.141 -h 00:01:02:03:04:05"},
}
```

```
/* Add a host name (dhcp option 12) for this host */
{"ipdhcps.netconf.11", "option add 10.0.0.141 12 yellow-host"},

/* Set up a boot file for yellow-host */
{"ipdhcps.netconf.12",
 "config set 10.0.0.141 boot-file /boot/yellow-host.boot"},

/* Set up a boot server for yellow-host to boot from */
{"ipdhcps.netconf.13",
 "config set 10.0.0.141 next-server-name black-server"},

/* Set the global default lease time to 600 seconds */
{"ipdhcps.netconf.14", "config set default default-lease-time 600"},

/* Set the global default ARP-cache timeout to 60 seconds */
{"ipdhcps.netconf.15", "option add default 35 60"},

...

/* Mark the end of the option list*/
{ IP_NULL, IP_NULL}
};
```

4.3.2 Configuring the Server with Shell Commands

You can configure the server at run time in two ways:

1. You can dynamically configure the server using DHCP shell commands. The shell commands are described in this section.
2. You can also configure the server by writing the `ipdhcps_start_hook()` routine to read a configuration file when the DHCP daemon starts up, see [4.4.1 The `ipdhcps_start_hook\(\)` Routine](#), p.66.

In either case, two sets of configuration parameters are available. The first set is made up of DHCP options specified in RFC 2132: *DHCP Options and BOOTP Vendor Extensions* (see [4.5 Setting Options in Shell Commands and API Routines](#), p.72, [Table 4-3](#)). The second set is made up of Wind River-specific configuration options (see [4.5 Setting Options in Shell Commands and API Routines](#), p.72, [Table 4-4](#)).

DHCP Server Shell Commands

DHCP server shell commands start with the key word **dhcps** and have the form:

dhcps *command subcommand* [0 or more arguments]

For example, the following command adds a new subnet with a specified network address and mask to the DHCP server's configuration database:

```
-> dhcpd subnet add 10.1.2.0 255.255.255.0
```

The seven DHCP server shell commands are described in the following sections:

dhcpd subnet

```
dhcpd subnet list
```

List all of the subnets in the configuration database.

```
dhcpd subnet add netAddress mask
```

Add a subnet with the specified network address and mask to the configuration database.

```
dhcpd subnet delete address
```

Delete the subnet with the specified network address from the configuration database.

dhcpd pool

```
dhcpd pool list subnet
```

List all of the address pools that are available on a specified subnet.

```
dhcpd pool add subnet firstAddress lastAddress
```

Add an address pool to a subnet.

```
dhcpd pool delete firstAddress lastAddress
```

Delete an address pool from a subnet.

dhcpd host

A network administrator assigns static IP addresses and identifiers to hosts in the configuration database.

```
dhcpd host list
```

List all of the hosts in the configuration database.

```
dhcpd host add hostIPAddress [-h linkLayerAddress | -i clientIdentifier]
```

Add the specified host to the configuration database. To specify the host, use the host's IP address and, optionally, a link-layer address or a client identifier:

linkLayerAddress

This is the host's hardware address "typed by the type of hardware to accommodate possible duplication of hardware addresses resulting from bit-ordering problems in a mixed-media..." (RFC 2131, section 2.1)

clientIdentifier

A client identifier, as defined in RFC 2131, section 2.

```
dhcps host delete hostIPaddress
```

Delete the specified host from the configuration database.

dhcps lease

```
dhcps lease list
```

List all of the entries in the lease database.

```
dhcps lease flush [ -s bindingState | -h linkLayerAddress | -i clientID | -a IPv4address ]
```

Delete one or more leases from the lease database, based on the following search criteria:

bindingState

Flush all leases in the specified state—one of the following:

- **active**
- **expired**
- **released**
- **abandoned**
- **bootp**

linkLayerAddress

The hardware address of the client whose lease is to be flushed, as entered in the **dhcps host add** command (see [dhcps host](#), p.63).

clientID

The client identifier of the client whose lease is to be flushed, as defined in RFC 2131, section 2.

IPv4address

The IP address of the client whose lease is to be flushed.

When you flush a lease, this deletes it from the lease database and frees the IP address associated with the lease, which DHCP can then reassign to a different host.

dhcps option

The **dhcps option** command applies to DHCP options specified in RFC 2132. For a table of these options as used with the Wind River DHCP server, see [4.5 Setting Options in Shell Commands and API Routines](#), p.72, [Table 4-3](#).

```
dhcps option list address
```

List the DHCP option values that apply to the specified host or subnet address. To list the server's default values for DHCP options, use **default** as the address.


```
dhcps option add address optionCode value
```

Add a DHCP option to the default set of options or to the options associated with a specified host address or subnet address. To add an option to the default set, use **default** as the address.

```
dhcps option delete address optionCode
```

Delete a DHCP network option from the default set of options or from the options associated with a specified host address or subnet address. To delete an option from the default set, use **default** as the address.

dhcps config

The **dhcps config** command applies to options that are specific to Wind River's DHCP implementation. For information on these options, see [4.5 Setting Options in Shell Commands and API Routines](#), p.72, [Table 4-4](#).

```
dhcps config list address
```

List the server's default set of **dhcps config** options or the options associated with a specific host address or subnet address. To list the default options, use **default** as the address.

```
dhcps config set address optionCode value
```

Set the value of a **dhcps config** option in either the server's default set of options or in the options associated with a specified host address or subnet address. To set an option in the server's default set, use **default** as the address.

```
dhcps config reset address optionCode
```

Reset the value of a DHCP config option. If you enter a host or subnet address for *address*, this resets the config option to the current value of the option in the server's default set of options. If you enter **default** for *address*, this resets the server's default value for the option to the value it had at system startup.

dhcps interface

```
dhcps interface list
```

List the interfaces that the DHCP server can use.

```
dhcps interface enable interfaceName
```

Allow the DHCP server to use the specified interface; that is, instruct the DHCP server to listen for DHCPDISCOVER messages that arrive on that interface.

```
dhcps interface disable interfaceName
```

Disable the use of the specified interface by the DHCP server.

4.4 Implementing Hook Routines for Initialization and Shutdown

If you want the DHCP server to do any custom processing of your choosing at startup or shutdown, you can implement this through the `ipdhcps_start_hook()` and `ipdhcps_stop_hook()` routines. For instance, you may want to restore the lease database at startup, and store this database to a file at shutdown. These hook routines are described in this section.

To use your own custom hook routines, you must complete the following steps:

1. Write your routine or routines, according to the instructions in [4.4.1 The `ipdhcps_start_hook\(\)` Routine](#), p.66 and [4.4.2 The `ipdhcps_stop_hook\(\)` Routine](#), p.70.
2. Set the configuration parameter `DHCPS_INSTALL_CALLBACK_HOOK` to `TRUE`. See [4.2 Including the DHCP Server in a Build](#), p.52.
3. Set either or both of the configuration parameters `DHCPS_START_CALLBACK_HOOK` and `DHCPS_STOP_CALLBACK_HOOK` to point to your custom hook routines. See [4.2 Including the DHCP Server in a Build](#), p.52.
4. Rebuild your target.

4.4.1 The `ipdhcps_start_hook()` Routine

The DHCPS daemon calls the `ipdhcps_start_hook()` routine at startup. The signature of `ipdhcps_start_hook()` is:

```
IP_GLOBAL int ipdhcps_start_hook (void)
```

This routine returns zero to indicate that initialization has succeeded (or one to indicate a failure)

Write `ipdhcps_start_hook()` to do any initialization you need to accomplish, which may include calling any of the DHCP server API. You can use this API to restore the lease database and to configure the server with global defaults, subnets, hosts, and options. For a list and brief descriptions of the routines in the API, see [4.4.3 DHCP Server API Routines](#), p.71. For detailed information about individual routines, see the API reference pages.

You can implement `ipdhcps_start_hook()` to configure the server by reading a configuration file at startup. For an example of this, see [Example 4-3](#). You could also call DHCP-server API routines with hard-coded arguments, however this is equivalent to statically configuring the server through entries in the

netconf_sysvar array entries in the **dhcps_config.c** file, which is simpler and less error prone.

Example 4-2 Restoring the Lease Database at Startup

The following implementation of **ipdhcps_start_hook()** assumes that the data for the lease database is in a file. It reads this file into a buffer and then calls **ipdhcps_lease_db_restore()**.

```
IP_GLOBAL int ipdhcps_start_hook (void)
{
    IP_FILE      fd;
    struct stat  statbuf;
    void *       buf;

    /* Get size of dump file */
    stat ("/opt/ipdhcps/dumpfile", &statbuf);

    /* Transfer data from file to a data buffer */
    fd = fopen ("/opt/ipdhcps/dumpfile", "r");
    buf = malloc (statbuf.st_size);
    fread (buf, statbuf.st_size, 1, fd);

    /* Restore lease database from file */
    ipdhcps_lease_db_restore (buf);

    fclose (fd);
    free (buf);

    return (0);
}
```

Example 4-3 Reading Configuration Values from a File

The following sample implementation of the **ipdhcps_start_hook()** routine reads a configuration file in which each line is a string that corresponds to a DHCP-server shell command (see *DHCP Server Shell Commands*, p.62) without the initial **dhcps** command identifier, as in the following example:

```
subnet add 10.1.0.0 255.255.0.0
```

The routine parses the command string using the **getOptServ()** routine (for more detailed information, see the **getOptServ()** reference page) and then calls the **ipdhcps_cmd_dhcp()** routine, which executes the command (for more information on the **ipdhcps_cmd_dhcp()** routine, see *The ipdhcps_cmd_dhcp() routine*, p.69).

```
IP_GLOBAL int ipdhcps_start_hook (void)
{
    IP_FILE *      fd          = IP_NULL;
    char *         cmdbuf      = IP_NULL;
    char           args[100];
```

```
const char * cfgfile    = "/home/dhcpServer/dhcps.conf";
const char * cmd        = "dhcps ";
int          argc;
char *       tempargv   = [106 + 2];
char **      argv       = tempargv;
int          ret        = -1;

if ((fd = fopen(cfgfile,"r")) == 0)
{
    logMsg ("DHCPD ERROR: Failed to open configuration file %s\n",
            cfgfile, 0, 0, 0, 0, 0);
    goto leave;
}

/*
 * The config file consists of command line arguments that are
 * understood by the DHCP server's shell tool
 */

/* Read the command lines one by one and execute */
while (fgets (args, sizeof (args), fd) != IP_NULL)
{
    /* Remove the trailing EOLN */
    args [strlen (args) - 1] = 0;

    /* Compose a complete command args */
    cmdbuf = malloc (strlen (args) + strlen (cmd) + 1);

    if (cmdbuf == IP_NULL)
    {
        logMsg ("DHCPD ERROR: failed to allocate buffer\n",
                0, 0, 0, 0, 0, 0);
        goto leave;
    }

    strcpy (cmdbuf, cmd);
    strcat (cmdbuf, args);

    if (getOptServ (cmdbuf, "ipdhcps_start_hook", &argc, argv, 106 + 2)
        != OK)
    {
        logMsg ("DHCPD ERROR: failed to parse command '%s'\n",
                args, 0, 0, 0, 0, 0);
        goto leave;
    }

    /* Let the dhcps command tool execute */
    if (ipdhcps_cmd_dhcps (argc, argv) != OK)
    {
        logMsg ("DHCPD ERROR: failed to execute command '%s'\n",
                args, 0, 0, 0, 0, 0);
        goto leave;
    }

    free (cmdbuf);
    cmdbuf = NULL;
}
```

```

    }

    ret = OK;

leave:

    if (fd)
        fclose (fd);

    if (cmdbuf)
        free (cmdbuf);

    return ret;
}

```

Sample Configuration File

```

subnet add 10.1.0.0 255.255.0.0
pool add 10.1.0.0 10.1.0.5 10.1.0.99
pool add 10.1.0.0 10.1.0.199 10.1.0.249
option add 10.1.0.0 15 green.net
option add 10.1.0.0 3 10.1.0.1
option add 10.1.0.0 5 10.1.0.2
subnet add 192.168.2.0 255.255.255.0
pool add 192.168.2.0 192.168.2.10 192.168.2.50
option add 192.168.2.0 15 blue.net
option add 192.168.2.0 3 192.168.2.1 192.168.2.2
option add 192.168.2.0 33 10.1.0.0 192.168.2.2
config set 192.168.2.0 max-lease-time 120
config set 192.168.2.0 default-lease-time 60
host add 10.1.0.142 -i \0Jack's lap top
option add 10.1.0.142 12 blue-host
host add 10.1.0.141 -h BE:60:13:F7:F8:0F
option add 10.1.0.141 12 yellow-host
config set 10.1.0.141 boot-file /boot/yellow-host.boot
config set 10.1.0.141 next-server-name black-server
config set default default-lease-time 600
config set default allow-bootp false
config set 10.1.0.0 do-icmp-address-check true
class add 10.1.0.0 first-class vendor first
class add default second-class vendor second
option add 10.1.0.0 35 10
option add 10.1.0.0 -c first-class 35 20
option add default 35 60
option add default -c second-class 35 40
config set 10.1.0.0 authorized-agents 10.1.0.2 10.1.0.3

```

The `ipdhcps_cmd_dhcpd()` routine

The `ipdhcps_cmd_dhcpd()` routine executes a DHCP-server shell command. The syntax for the `ipdhcps_cmd_dhcpd()` routine is:

```
IP_GLOBAL int ipdhcps_cmd_dhcpd (int argc, char ** argv)
```

where:

argc is the number of strings in the array pointed to by the *argv* parameter.

argv is an array of strings that constitute the elements of a DHCP-server shell command, without the initial **dhcps** identifier. The required elements in the array are:

```
command subcommand { 0 or more arguments }
```

For example, for the **subnet add** shell command, *argv* could contain the following strings:

```
"subnet" "add" "10.1.0.0" "255.255.0.0"
```

The **ipdhcps_cmd_dhcps()** routine returns 0 on success.

4.4.2 The **ipdhcps_stop_hook()** Routine

The DHCP daemon automatically calls the **ipdhcps_stop_hook()** routine immediately before the server exits. The signature of **ipdhcps_stop_hook()** is:

```
IP_GLOBAL int ipdhcps_stop_hook (void)
```

This routine returns zero to indicate that the hook routine has succeeded (or one to indicate a failure)

You can write an implementation of the **ipdhcps_stop_hook()** routine that does any shutdown and cleanup processing necessary to your application, including dumping the lease database using the **ipdhcps_lease_db_dump()** API.

Example 4-4 Dumping the Lease Database at Shutdown

The following implementation of **ipdhcps_stop_hook()** reads the contents of the lease database into a buffer using the **ipdhcps_lease_db_dump()** API and then writes that buffer to a file:

```
IP_GLOBAL int ipdhcps_stop_hook ()
{
    void *    buf;
    IP_FILE * fd;
    int       n;

    /* Dump the lease database */
    n = ipdhcps_lease_db_dump (&buf);

    /* Write it to a file */
    fd = fopen ("/opt/ipdhcps/dumpfile", "w+");
    fwrite (buf, n, 1, fd);

    fclose (fd);
    free (buf);
}
```

```
return (0);  
}
```

4.4.3 DHCP Server API Routines

Table 4-2 lists the API routines available to DHCP server applications and gives a brief description of each routine. For more detailed information, see the reference pages for these routines.

Table 4-2 DHCP-Server APIs

Routine	Description
ipdhcps_subnet_add()	Add a new subnet to the configuration database.
ipdhcps_subnet_delete()	Delete a subnet from the configuration database.
ipdhcps_pool_add()	Add a new address pool to an existing subnet.
ipdhcps_pool_delete()	Delete an existing address pool from a subnet.
ipdhcps_host_add()	Add a new host to the configuration database.
ipdhcps_host_delete()	Delete a host from the configuration database.
ipdhcps_dhcp_option_add()	Add a set of DHCP options to a subnet or host.
ipdhcps_dhcp_option_delete()	Delete a DHCP option.
ipdhcps_config_option_set()	Set configuration options on a subnet or host.
ipdhcps_config_option_reset()	Reset a configuration option to its default value.
ipdhcps_interface_status_set()	Set the status of an interface.
ipdhcps_lease_db_dump()	Dump entries in the lease database to a buffer.
ipdhcps_lease_db_restore()	Initialize the lease database.

4.5 Setting Options in Shell Commands and API Routines

You can set standard DHCP options and Wind River-specific options statically in the `ipdhcps_config.c` file. In this case, use numerical codes for individual options, as given in [Table 4-3](#) (from RFC 2132) and [Table 4-4](#).

You can also set DHCP options through shell commands. In this case, you enter the name of an option as specifically required by the shell. For example, for option code 1, you need to enter “subnet mask” as the option name (see [Table 4-3](#)).

Finally, if you set DHCP option values through DHCP server API routines, instead of directly entering a numerical option value, enter a macro value for the code. For example, for option code 1, use `IPDHCP_OPTCODE_SUBNET_MASK` (see [Table 4-3](#) and [Table 4-4](#) for lists of these macros).

4.5.1 Using Standard DHCP Options in Shell Commands and APIs

The following table lists the standard DHCP options available in Wind River DHCP, ordered by code number, and gives their macro names and shell names as required in API routines and shell commands.

Table 4-3 **DHCP Options As Used in Shell Commands and APIs**

Code	Macro for Configuration Option, and Option Name in Shell Command	Option Type
1	<code>IPDHCP_OPTCODE_SUBNET_MASK</code> subnet mask	IPv4 address
2	<code>IPDHCP_OPTCODE_TIME_OFFSET</code> time offset	32-bits
3	<code>IPDHCP_OPTCODE_ROUTERS</code> router(s)	list of IPv4 addresses
4	<code>IPDHCP_OPTCODE_TIME_SERVERS</code> time server(s)	list of IPv4 addresses
5	<code>IPDHCP_OPTCODE_NAME_SERVERS</code> name server(s)	list of IPv4 addresses
6	<code>IPDHCP_OPTCODE_DOMAIN_NAME_SERVERS</code> domain name server(s)	list of IPv4 addresses

Table 4-3 **DHCP Options As Used in Shell Commands and APIs** (cont'd)

Code	Macro for Configuration Option, and Option Name in Shell Command	Option Type
7	IPDHCPD_OPTS_LOG_SERVERS log server(s)	list of IPv4 addresses
8	IPDHCPD_OPTS_COOKIE_SERVERS cookie server(s)	list of IPv4 addresses
9	IPDHCPD_OPTS_LPR_SERVERS lpr server(s)	list of IPv4 addresses
10	IPDHCPD_OPTS_IMPRESS_SERVERS impress server(s)	list of IPv4 addresses
11	IPDHCPD_OPTS_RESOURCE_LOCATION_SERVERS resource location server(s)	list of IPv4 addresses
12	IPDHCPD_OPTS_HOST_NAME host name	string
13	IPDHCPD_OPTS_BOOT_SIZE boot file size	16-bits
14	IPDHCPD_OPTS_MERIT_DUMP merit dump path	string
15	IPDHCPD_OPTS_DOMAIN_NAME domain name	string
16	IPDHCPD_OPTS_SWAP_SERVER swap server(s)	IPv4 address
17	IPDHCPD_OPTS_ROOT_PATH root path	string
18	IPDHCPD_OPTS_EXTENSIONS_PATH extensions path	string
19	IPDHCPD_OPTS_IP_FORWARDING ip forwarding	boolean
20	IPDHCPD_OPTS_NON_LOCAL_SOURCE_ROUTING non local source routing	boolean

Table 4-3 **DHCP Options As Used in Shell Commands and APIs** (cont'd)

Code	Macro for Configuration Option, and Option Name in Shell Command	Option Type
7	IPDHCPD_OPTCODE_LOG_SERVERS log server(s)	list of IPv4 addresses
8	IPDHCPD_OPTCODE_COOKIE_SERVERS cookie server(s)	list of IPv4 addresses
9	IPDHCPD_OPTCODE_LPR_SERVERS lpr server(s)	list of IPv4 addresses
10	IPDHCPD_OPTCODE_IMPRESS_SERVERS impress server(s)	list of IPv4 addresses
11	IPDHCPD_OPTCODE_RESOURCE_LOCATION_SERVERS resource location server(s)	list of IPv4 addresses
12	IPDHCPD_OPTCODE_HOST_NAME host name	string
13	IPDHCPD_OPTCODE_BOOT_SIZE boot file size	16-bits
14	IPDHCPD_OPTCODE_MERIT_DUMP merit dump path	string
15	IPDHCPD_OPTCODE_DOMAIN_NAME domain name	string
16	IPDHCPD_OPTCODE_SWAP_SERVER swap server(s)	IPv4 address
17	IPDHCPD_OPTCODE_ROOT_PATH root path	string
18	IPDHCPD_OPTCODE_EXTENSIONS_PATH extensions path	string
19	IPDHCPD_OPTCODE_IP_FORWARDING ip forwarding	boolean
20	IPDHCPD_OPTCODE_NON_LOCAL_SOURCE_ROUTING non local source routing	boolean

Table 4-3 **DHCP Options As Used in Shell Commands and APIs** (cont'd)

Code	Macro for Configuration Option, and Option Name in Shell Command	Option Type
21	IPDHCPS_OPTCODE_POLICY_FILTER policy filter	list of IPv4 address pairs
22	IPDHCPS_OPTCODE_MAX_DGRAM_REASSEMBLY max datagram reassembly	16-bits
23	IPDHCPS_OPTCODE_DEFAULT_IP_TTL ip time to live	8-bits
24	IPDHCPS_OPTCODE_PATH_MTU_AGING_TIMEOUT path mtu aging timeout	32-bits
25	IPDHCPS_OPTCODE_PATH_MTU_PLATEAU_TABLE path mtu plateau table	list of 16-bit values
26	IPDHCPS_OPTCODE_INTERFACE_MTU interface mtu	16-bits
27	IPDHCPS_OPTCODE_ALL_SUBNETS_LOCAL all subnets local	boolean
28	IPDHCPS_OPTCODE_BROADCAST_ADDRESS broadcast address	IPv4 address
29	IPDHCPS_OPTCODE_PERFORM_MASK_DISCOVERY perform mask discovery	boolean
30	IPDHCPS_OPTCODE_MASK_SUPPLIER subnet mask supplier	boolean
31	IPDHCPS_OPTCODE_ROUTER_DISCOVERY perform router discovery	boolean
32	IPDHCPS_OPTCODE_ROUTER_SOLICITATION_ADDRESS router solicitation address	IPv4 address
33	IPDHCPS_OPTCODE_STATIC_ROUTES static routes	list of IPv4 address pairs
34	IPDHCPS_OPTCODE_TRAILER_ENCAPSULATION trailer encapsulation	boolean

Table 4-3 **DHCP Options As Used in Shell Commands and APIs** (cont'd)

Code	Macro for Configuration Option, and Option Name in Shell Command	Option Type
7	IPDHCPD_OPTCODE_LOG_SERVERS log server(s)	list of IPv4 addresses
8	IPDHCPD_OPTCODE_COOKIE_SERVERS cookie server(s)	list of IPv4 addresses
9	IPDHCPD_OPTCODE_LPR_SERVERS lpr server(s)	list of IPv4 addresses
10	IPDHCPD_OPTCODE_IMPRESS_SERVERS impress server(s)	list of IPv4 addresses
11	IPDHCPD_OPTCODE_RESOURCE_LOCATION_SERVERS resource location server(s)	list of IPv4 addresses
12	IPDHCPD_OPTCODE_HOST_NAME host name	string
13	IPDHCPD_OPTCODE_BOOT_SIZE boot file size	16-bits
14	IPDHCPD_OPTCODE_MERIT_DUMP merit dump path	string
15	IPDHCPD_OPTCODE_DOMAIN_NAME domain name	string
16	IPDHCPD_OPTCODE_SWAP_SERVER swap server(s)	IPv4 address
17	IPDHCPD_OPTCODE_ROOT_PATH root path	string
18	IPDHCPD_OPTCODE_EXTENSIONS_PATH extensions path	string
19	IPDHCPD_OPTCODE_IP_FORWARDING ip forwarding	boolean
20	IPDHCPD_OPTCODE_NON_LOCAL_SOURCE_ROUTING non local source routing	boolean

Table 4-3 **DHCP Options As Used in Shell Commands and APIs** (cont'd)

Code	Macro for Configuration Option, and Option Name in Shell Command	Option Type
21	IPDHCPS_OPTCODE_POLICY_FILTER policy filter	list of IPv4 address pairs
22	IPDHCPS_OPTCODE_MAX_DGRAM_REASSEMBLY max datagram reassembly	16-bits
23	IPDHCPS_OPTCODE_DEFAULT_IP_TTL ip time to live	8-bits
24	IPDHCPS_OPTCODE_PATH_MTU_AGING_TIMEOUT path mtu aging timeout	32-bits
25	IPDHCPS_OPTCODE_PATH_MTU_PLATEAU_TABLE path mtu plateau table	list of 16-bit values
26	IPDHCPS_OPTCODE_INTERFACE_MTU interface mtu	16-bits
27	IPDHCPS_OPTCODE_ALL_SUBNETS_LOCAL all subnets local	boolean
28	IPDHCPS_OPTCODE_BROADCAST_ADDRESS broadcast address	IPv4 address
29	IPDHCPS_OPTCODE_PERFORM_MASK_DISCOVERY perform mask discovery	boolean
30	IPDHCPS_OPTCODE_MASK_SUPPLIER subnet mask supplier	boolean
31	IPDHCPS_OPTCODE_ROUTER_DISCOVERY perform router discovery	boolean
32	IPDHCPS_OPTCODE_ROUTER_SOLICITATION_ADDRESS router solicitation address	IPv4 address
33	IPDHCPS_OPTCODE_STATIC_ROUTES static routes	list of IPv4 address pairs
34	IPDHCPS_OPTCODE_TRAILER_ENCAPSULATION trailer encapsulation	boolean

Table 4-3 **DHCP Options As Used in Shell Commands and APIs** (cont'd)

Code	Macro for Configuration Option, and Option Name in Shell Command	Option Type
35	IPDHCPD_OPTCODE_ARP_CACHE_TIMEOUT arp cache timeout	32-bits
36	IPDHCPD_OPTCODE_ETHERNET_802_3_ENCAPSULATION ethernet encapsulation	boolean
37	IPDHCPD_OPTCODE_DEFAULT_TCP_TTL default tcp time to live	8-bits
38	IPDHCPD_OPTCODE_TCP_KEEPALIVE_INTERVAL tcp keep alive interval	32-bits
39	IPDHCPD_OPTCODE_TCP_KEEPALIVE_GARBAGE tcp keep alive garbage	boolean
40	IPDHCPD_OPTCODE_NIS_DOMAIN nis domain	string
41	IPDHCPD_OPTCODE_NIS_SERVERS nis server(s)	list of IPv4 addresses
42	IPDHCPD_OPTCODE_NTP_SERVERS ntp server(s)	list of IPv4 addresses
43	IPDHCPD_OPTCODE_VENDOR_ENCAPSULATED_OPTIONS vendor encapsulated options	string
44	IPDHCPD_OPTCODE_NETBIOS_NAME_SERVERS netbios name server(s)	list of IPv4 addresses
45	IPDHCPD_OPTCODE_NETBIOS_DD_SERVER netbios dgram distr server(s)	list of IPv4 addresses
46	IPDHCPD_OPTCODE_NETBIOS_NODE_TYPE netbios node type	8-bits
47	IPDHCPD_OPTCODE_NETBIOS_SCOPE netbios scope	string
48	IPDHCPD_OPTCODE_FONT_SERVERS X font server(s)	list of IPv4 addresses

Table 4-3 **DHCP Options As Used in Shell Commands and APIs** (cont'd)

Code	Macro for Configuration Option, and Option Name in Shell Command	Option Type
49	IPDHCPD_OPTS_DISPLAY_MANAGER X display manager	list of IPv4 addresses
58	—	—
59	—	—
61	IPDHCPD_OPTS_CLIENT_IDENTIFIER —	—
62	IPDHCPD_OPTS_NWIP_DOMAIN —	—
64	IPDHCPD_OPTS_NISPLUS_DOMAIN nis+ domain	string
65	IPDHCPD_OPTS_NISPLUS_SERVERS nis+ server(s)	list of IPv4 addresses
68	IPDHCPD_OPTS_HOME_AGENTS mobile ip home agent	list of IPv4 addresses
69	IPDHCPD_OPTS_SMTP_SERVERS smtp server(s)	list of IPv4 addresses
70	IPDHCPD_OPTS_POP3_SERVERS pop3 server(s)	list of IPv4 addresses
71	IPDHCPD_OPTS_NNTP_SERVERS nntp server(s)	list of IPv4 addresses
72	IPDHCPD_OPTS_WWW_SERVERS www server(s)	list of IPv4 addresses
73	IPDHCPD_OPTS_FINGER_SERVERS finger server(s)	list of IPv4 addresses
74	IPDHCPD_OPTS_IRC_SERVERS irc server(s)	list of IPv4 addresses

Table 4-3 **DHCP Options As Used in Shell Commands and APIs** (cont'd)

Code	Macro for Configuration Option, and Option Name in Shell Command	Option Type
75	IPDHCPD_OPTCODE_STREETTALK_SERVERS street talk server(s)	list of IPv4 addresses
76	IPDHCPD_OPTCODE_STDA_SERVERS stda server(s)	list of IPv4 addresses

4.5.2 Using Wind River-Specific Options in Shell Commands and APIs

Table 4-4 lists the configuration options specific to the Wind River implementation of DHCP, and gives their the macro names and shell names as required in APIs and shell commands.

Table 4-4 **Configuration Options Specific to Wind River DHCP**

Code	Macro for Configuration Option, and Option Name in Shell Command	Description	Option Type
0x8006	IPDHCPD_CONFCODE_ALLOW_BOOTP allow-bootp	Enable the server to accept requests from BOOTP clients. You can also set this option as a build parameter.	boolean
0x8007	IPDHCPD_CONFCODE_ALLOW_DYNAMIC_BOOTP allow-dynamic-bootp	Enable the server to accept dynamic leases for BOOTP. You can also set this option as a build parameter.	boolean
0x800a	IPDHCPD_CONFCODE_BOOT_FILE boot-file	Set the name of the boot file.	string
0x8001	IPDHCPD_CONFCODE_LEASE_TIME_DFLT default-lease-time	Set the default lease time for clients, in seconds. You can also set this option as a build parameter. Give the option in host byte order.	integer

Table 4-4 Configuration Options Specific to Wind River DHCP (cont'd)

Code	Macro for Configuration Option, and Option Name in Shell Command	Description	Option Type
0x8002	IPDHCPD_CONFCONF_CODE_LEASE_TIME_MAX max-lease-time	Set the maximum lease time for clients, in seconds. You can also set this option as a build parameter. Give the option in host byte order.	integer
0x8003	IPDHCPD_CONFCONF_CODE_LEASE_TIME_MIN min-lease-time	Set the minimum lease time for clients, in seconds. You can also set this option as a build parameter. Give the option in host byte order.	integer
0x8008	IPDHCPD_CONFCONF_CODE_NEXT_SERVER_NAME next-server-name	Give the name of the next server in the boot chain.	string
0x8009	IPDHCPD_CONFCONF_CODE_NEXT_SERVER_IP next-server-ip	Give the IPv4 address of the next server in the boot chain, in network byte order.	four bytes
0x8005	IPDHCPD_CONFCONF_CODE_REBINDING_TIME rebinding-time	Set the lease rebinding time for clients, in seconds. You can also set this option as a build parameter. Give this option in host byte order.	integer
0x8004	IPDHCPD_CONFCONF_CODE_RENEWAL_TIME renewal-time	Set the lease renewal time for clients, in seconds. You can also set this option as a build parameter. Give the option in host byte order.	integer

5

Wind River DHCP: Relay Agent

- 5.1 Introduction 83
- 5.2 Including the DHCP Relay Agent in a Build 85
- 5.3 Configuring the Relay Agent with the `ipdhcpr_netconf_sysvar` Array 87
- 5.4 Using Shell Commands 88
- 5.5 Implementing the `ipdhcpr_start_hook()` Routine 89

5.1 Introduction

This chapter describes the Wind River DHCP relay agent. For a general overview of DHCP, see [3. *Wind River DHCP and DHCPv6: Overview*](#).

Relay Agent Overview

A *relay agent* is an Internet host or router that passes DHCP messages between DHCP clients and DHCP servers. There can be multiple relay agents between a client and a server. A DHCP relay agent is also a BOOTP relay agent, as specified in RFCs 951 and 1542. The Wind River DHCP relay agent runs as a daemon process that acts as a stateless message forwarder between a DHCP client and a DHCP server. The relay agent examines each packet in a DHCP or BOOTP message.

Packets Sent to a DHCP Server

When the relay agent receives a message from a DHCP client, it forwards the message on to all the DHCP servers that the relay agent knows about.

The DHCP/BOOTP header in DHCP packets contains an “agent” field that is meant to contain the IP address of the first relay agent that handles a request by a client. The address indicates the network segment on which the client is located. Before forwarding a DHCP packet to a server, the relay agent checks the DHCP/BOOTP header to see if it already contains an agent address. If it does not, the relay agent assumes that it is the first agent to see the message and it fills the field with the address of the interface on which it received the message, before passing the message on.

Packets Sent to the Client

If a relay agent receives a packet intended for a DHCP client, the relay agent checks the “agent” field and takes one of the following actions:

- If the agent field contains the address of one of its own interfaces, the agent relays the packet directly to the client.
- If the agent field contains an address that does not belong to one of its interfaces but matches the subnet of an other relay agent, the relay agent forwards the packet to the next agent.
- If the address in the agent field does not match a known agent or one of its own interfaces, the relay agent broadcasts the message on the subnet identified in the address.

Conformance to Standards

The Wind River DHCP relay agent implements all relay agent-specific features of RFC 951: *Bootstrap Protocol*, and RFC 1542: *Clarifications and Extensions for the Bootstrap Protocol*.

5.2 Including the DHCP Relay Agent in a Build

To include the DHCP relay agent in a VxWorks build, you must include the **DHCP Relay Agent** (INCLUDE_IPDHCP) build component.

Build Configuration Parameters

The **DHCP Relay Agent** (INCLUDE_IPDHCP) build component has a number of configuration parameters. [Table 5-1](#) lists and describes each parameter. See [3.1.3 Build Configuration Parameters and sysvars](#), p.46 for more information on setting these parameters.

Table 5-1 DHCP Relay-Agent Build Parameters

Workbench Description, Parameter Name, and sysvar	Default Value & Data Type
UDP port used by the dhcp server DHCPR_SERVER_PORT ipdhcpr.ServerPort The UDP port on which the DHCP server receives messages. Enter the value as a quoted string.	"67" char *
UDP port used by the dhcp/bootp clients DHCPR_CLIENT_PORT ipdhcpr.ClientPort The UDP port on which the DHCP client receives messages. Enter the value as a quoted string.	"68" char *
Max dhcp relay packet size DHCPR_MAX_PKT_SIZE ipdhcpr.PacketSize The maximum packet size, in bytes, the DHCP relay agent will use when sending and receiving packets. The size may never be less than 576 bytes, which is the standard packet size for DHCP.	"576" char *

Table 5-1 **DHCP Relay-Agent Build Parameters** (cont'd)

Workbench Description, Parameter Name, and sysvar	Default Value & Data Type
Maximum number of hops DHCPR_HOPS_THRESHOLD ipdhcpr.HopsThreshold The maximum number of relay agents allowed between a DHCP client and a DHCP server. The agent silently drops packets with a hop count greater than this setting.	"3" char *
DHCP relay network pre-configuration DHCPR_NETCONF_SYSVAR sysvar: None. Either "true" or "false". If "true", you can statically configure the relay agent by entering configuration values in the ipdhcpr_netconf_sysvar array in the ipdhcpr_config.c file. For information on entering values into the array, see 5.3 Configuring the Relay Agent with the ipdhcpr_netconf_sysvar Array , p.87.	NULL void *
Install dhcp relay callback routine DHCPR_INSTALL_CALLBACK_HOOK sysvar: None. Set this to TRUE if you implement the ipdhcpr_start_hook() routine (and also set the following parameter to point to your implementation of that routine). For information on ipdhcpr_start_hook() , see 5.5 Implementing the ipdhcpr_start_hook() Routine , p.89.	FALSE BOOL
DHCP relay startup callback routine DHCPR_START_CALLBACK_HOOK sysvar: None. If you set this function pointer and set the Install dhcp relay callback routine parameter to TRUE, the DHCP relay agent calls this routine—your implementation of the ipdhcpr_start_hook() routine—when it starts. For information on ipdhcpr_start_hook() , see 5.5 Implementing the ipdhcpr_start_hook() Routine , p.89.	NULL funcptr

5.3 Configuring the Relay Agent with the `ipdhcpr_netconf_sysvar` Array

You can statically configure the relay agent by entering commands and parameter values in the `ipdhcpr_netconf_sysvar` array in the `ipdhcpr_config.c` file. For a list of the commands, see [Setting Commands in the `ipdhcpr_netconf_sysvar` Array](#), p.87. The path to `ipdhcpr_config.c` is:

`installDir/components/ip_net2-6.n/osconfig/vxworks/src/ipnet/ipdhcpr_config.c`

With the exception of the last entry in the array, entries have the following form:

```
{"ipdhcpr.netconf.index", "command action"}
```

In these entries, replace *index*, *command*, and *action* as follows:

index is an index into the `ipdhcpr_netconf_sysvar` array, and

command is an expression followed by an *action*, such as **add**, **enable**, or **disable**, as in the command **server add**, which is used to add a DHCP server to the relay agent's list of DHCP servers.

The last entry in the array must always be:

```
{IP_NULL, IP_NULL}
```

In the following example the first element in the `ipdhcpr_netconf_sysvar` array adds DHCP server 10.1.0.2 to the relay agent's list of DHCP servers:

```
IP_STATIC IP_CONST Ipcom_sysvar ipdhcpr_netconf_sysvar[] =
{
    {"ipdhcpr.netconf.00", "server add 10.1.0.2"},
    ...
    {IP_NULL, IP_NULL}
};
```

Setting Commands in the `ipdhcpr_netconf_sysvar` Array

You can use the following three commands in the DHCP relay agent configuration array:

```
server add address
```

Add a DHCP server to the relay agent's list of DHCP servers.

```
interface enable interfaceName
```

Enable an interface for DHCP.

```
interface disable interfaceName
```

Disable an interface for DHCP. By default, all interfaces are enabled for DHCP. You can use this command to disable specific interfaces for DHCP.

Example 5-1 **Sample ipdhcpr_netconf_sysvar Array**

The following example shows sample entries in an **ipdhcpr_netconf_sysvar** array:

```
IP_STATIC IP_CONST Ipcom_sysvar ipdhcpr_netconf_sysvar[] =
{
    /* Add DHCP server 10.1.0.2 */
    {"ipdhcpr.netconf.00", "server add 10.1.0.2"},

    /* Add DHCP server 192.168.1.1 */
    {"ipdhcpr.netconf.00", "server add 192.168.1.1"},

    /* Enable interface eth1 for DHCP */
    {"ipdhcpr.netconf.00", "interface enable eth1"},

    /* Mark the end of the array */
    {IP_NULL, IP_NULL}
};
```

5.4 Using Shell Commands

The relay agent provides two shell commands, the **dhcpr server** shell command and the **dhcpr interface** shell command.

Use the **dhcpr server** shell command to list all DHCP servers known to the relay agent, to add a server to the relay agent's list of servers, or to delete a server from this list.

Use the **dhcpr interface** shell command to list all interfaces enabled for DHCP, or to enable or disable individual interfaces for DHCP.

The **dhcpr server** and **dhcpr interface** shell commands are as follows:

```
dhcpr server list
```

List all DHCP servers known to the relay agent.

```
dhcpr server add address
```

Add a DHCP server to the relay agent's list of DHCP servers.

```
dhcpr server delete address
```

Delete a DHCP server from the relay agent's list of DHCP servers.

```
dhcpr interface list
```

List all interfaces enabled for DHCP on the relay agent.


```
dhcpr interface enable interfaceName  
    Enable the specified interface for DHCP.
```

```
dhcpr interface disable interfaceName  
    Disable the specified interface for DHCP.
```

By default, the relay agent enables all running interfaces for DHCP at startup (aside from the loopback interface and those interfaces, such as a PPP interface, that are not capable of broadcasting messages).

5

5.5 Implementing the `ipdhcpr_start_hook()` Routine

If you want to do some custom initialization at DHCP relay-agent startup, you can implement this in the form of an `ipdhcpr_start_hook()` initialization routine that the relay-agent will call when it starts. For instance, you can implement `ipdhcpr_start_hook()` to configure the relay agent by reading a configuration file at startup. The following section includes an example of this.

The signature of `ipdhcpr_start_hook()` is:

```
IP_GLOBAL int ipdhcpr_start_hook (void);
```

This routine returns zero to indicate that initialization has succeeded (or one to indicate a failure)

You can implement `ipdhcpr_start_hook()` to call any DHCP relay-agent API routine. For a list and brief descriptions of the API routines, see [5.5.1 DHCP Relay Agent API Routines](#), p.91. For detailed information about individual API routines, see their reference pages.

To enable your implementation of the `ipdhcpr_start_hook()` routine, you must set the **Install dhcp relay callback routine** and **DHCP relay callback startup routine** configuration parameters appropriately. See [Table 5-1](#).

Example 5-2 Reading Configuration Values from a File

The following is a sample implementation of the `ipdhcpr_start_hook()` routine that configures the relay-agent based on the contents of a configuration file, and a sample configuration file for this hook routine to operate on.

```
IP_GLOBAL int ipdhcpr_start_hook (void)  
{  
    IP_FILE *    fd          = IP_NULL;
```

```
char *      cmdbuf      = IP_NULL;
char      args[100];
const char * cfgfile    = "/home/dhcpRelay/dhcpr.conf";
const char * cmd        = "dhcpr ";
int        argc;
char *      tempargv    = [106 + 2];
char **     argv        = tempargv;
int         ret         = -1;

if ((fd = fopen (cfgfile,"r")) == 0)
{
    logMsg ("DHCP_RELAY ERROR: Failed to open configuration file %s\n",
            cfgfile, 0, 0, 0, 0, 0);
    goto leave;
}

/*
 * The config file consists of command line arguments that are
 * understood by the relay agent's shell tool
 */

/* Read the command lines one by one and execute */
while (fgets (args, sizeof (args), fd) != IP_NULL)
{
    /* Remove the trailing EOLN */
    args [strlen (args) - 1] = 0;

    /* Compose a complete command args */
    cmdbuf = malloc (strlen (args) + strlen (cmd) + 1);

    if (cmdbuf == IP_NULL)
    {
        logMsg ("DHCP_RELAY ERROR: failed to allocate buffer\n",
                0, 0, 0, 0, 0, 0);
        goto leave;
    }

    strcpy (cmdbuf, cmd);
    strcat (cmdbuf, args);

    if (getOptServ (cmdbuf, "ipdhcpr_start_hook", &argc, argv, 106 + 2)
        != OK)
    {
        logMsg ("DHCP_RELAY ERROR: failed to parse command '%s'\n",
                args, 0, 0, 0, 0, 0);
        goto leave;
    }

    /* Let the relay agent's command tool execute */
    if (ipdhcpr_cmd_dhcpr (argc, argv) != OK)
    {
        logMsg ("DHCP_RELAY ERROR: failed to execute command '%s'\n",
                args, 0, 0, 0, 0, 0);
        goto leave;
    }
}
```

```
        free (cmdbuf);
        cmdbuf = NULL;
    }

    ret = OK;

leave:

    if (fd)
        fclose (fd);

    if (cmdbuf)
        free (cmdbuf);

    return ret;
}
```

Sample Configuration File

```
server add 10.1.0.2
server add 192.168.1.1
interface enable eth1
```

5.5.1 DHCP Relay Agent API Routines

There are four API routines available to DHCP relay-agent applications:

`ipdhcpr_server_add()`

Add a new server address to the DHCP relay agent.

`ipdhcpr_server_delete()`

Delete a server address.

`ipdhcpr_interface_status_set()`

Set the status of an interface on the relay agent.

`ipdhcpr_start_hook()`

This is a callback for start-up operations.

For more detailed information, see the API reference pages for these routines.

6

Wind River DHCP: Client

- 6.1 Introduction 93
- 6.2 Including the DHCP Client in a Build 95
- 6.3 Using Shell Commands 100
- 6.4 Implementing the `ipdhcpc_option_callback()` Routine 101

6.1 Introduction

This chapter describes the Wind River DHCPv4 client. For a general overview of Wind River DHCP, see [3. *Wind River DHCP and DHCPv6: Overview*](#).

DHCPv4 Client Overview

A Wind River DHCP client obtains an IP address and additional configuration information from a DHCP server.

When a DHCP client starts, it scans through all available interfaces and checks the interface flags on each one. If the interface is up (`IP_IFF_UP` flag is set) and its DHCP flag (`IP_IFF_X_DHCPRUNNING`) is set, the client opens a DHCP session over it.

The client initiates a DHCP session on an interface by broadcasting a **DHCPDISCOVER** message over the interface. It broadcasts the message repeatedly until it gets a response from a DHCP server. The server's response message

contains an IP address for the client and a lease time that determines how long the address can be used. The client configures the interface with the address and keeps track of its lease time. The client contacts the server for lease renewal before the lease expires.

The response message from a DHCP server typically contains a number of configuration parameters (DHCP options) for the client, such as an interface subnet mask, an interface broadcast address, a default gateway, DNS servers, a DNS domain, and vendor-specific options. For each option, before it does anything else, the client calls `ipdhcpc_option_callback()`, a callback routine that you can implement to perform operations on DHCP options before any further processing by the client (see [6.4 Implementing the ipdhcpc_option_callback\(\) Routine](#), p.101).

6.1.1 Conformance to Standards

The Wind River DHCP client implements client portions of RFC 2131: *Dynamic Host Configuration Protocol*, and recognizes options and vendor extensions in RFC 2132: *DHCP Options and BOOTP Vendor Extensions*, and other specifications. In some cases, the client explicitly handles an option. In other cases, the client has no pre-existing implementation for handling an option, but allows you to implement operations for handling the option in the `ipdhcpc_option_callback()` routine (see [6.4 Implementing the ipdhcpc_option_callback\(\) Routine](#), p.101).

Implementation of DHCP Options

[Table 6-1](#) lists the DHCP options that the client handles directly.

Table 6-1 DHCP Options Implemented in the DHCP Client

Option Code	Macro Name
0	IPDHCPC_OPTCODE_PAD
1	IPDHCPC_OPTCODE_SUBNET_MASK
3	IPDHCPC_OPTCODE_ROUTERS
6	IPDHCPC_OPTCODE_DOMAIN_NAME_SERVERS
15	IPDHCPC_OPTCODE_DOMAIN_NAME
42	IPDHCPC_OPTCODE_NTP_SERVERS
51	IPDHCPC_OPTCODE_DHCP_LEASE_TIME

Table 6-1 DHCP Options Implemented in the DHCP Client (cont'd)

Option Code	Macro Name
53	IPDHCPC_OPTCODE_DHCP_MESSAGE_TYPE
54	IPDHCPC_OPTCODE_DHCP_SERVER_IDENTIFIER
55	IPDHCPC_OPTCODE_DHCP_PARAMETER_REQUEST_LIST
56	IPDHCPC_OPTCODE_DHCP_MESSAGE
57	IPDHCPC_OPTCODE_DHCP_MAX_MESSAGE_SIZE
58	IPDHCPC_OPTCODE_DHCP_RENEWAL_TIME
59	IPDHCPC_OPTCODE_DHCP_REBINDING_TIME
61	IPDHCPC_OPTCODE_DHCP_CLIENT_IDENTIFIER

If you want the DHCP client to handle options that are not listed in [Table 6-1](#), you must handle them in the `ipdhcpc_option_callback()` routine (see [6.4 Implementing the ipdhcpc_option_callback\(\) Routine](#), p.101).

6.2 Including the DHCP Client in a Build

To include the DHCP client in a VxWorks build, include the **DHCP Client** (`INCLUDE_IPDHCPC`) build component.



NOTE: An earlier version of DHCP for VxWorks allowed you to instruct the DHCP client not to allow the server to set its IP address, but only certain other network information. A client using DHCP in this way would set its IP address using some other method. The current version of DHCP does not allow this.

Build Configuration Parameters

The **DHCP Client** (`INCLUDE_IPDHCPC`) build component provides a number of configuration parameters. [Table 6-2](#) lists and describes each parameter. See [3.1.3 Build Configuration Parameters and sysvars](#), p.46 for more information on setting these parameters.

Table 6-2 **DHCP-Client Build Parameters**

Workbench Description, Parameter Name, and sysvar	Default Value & Data Type
DHCP client port DHCPC_CLIENT_PORT ipdhcpc.client_port <p>The port on which a DHCP client receives messages from the DHCP server. Enter the value as a quoted string.</p>	"68" char *
DHCP server port DHCPC_SERVER_PORT ipdhcpc.server_port <p>The DHCP server port to which the client sends messages. Enter the value as a quoted string.</p>	"67" char *
Global list of requested dhcp options DHCPC_REQ_OPTS ipdhcpc.requested_options <p>Specifies a comma-delimited list of DHCP option numbers (see RFC 2132) that the DHCP client wants the DHCP server to provide. For example, the following entry asks the server to provide a subnet mask, IP addresses for domain name servers, a broadcast address, and the IP address of an NTP server: "1,6,28,42".</p>	NULL char *
Global client identifier DHCPC_CLIENT_ID ipdhcpc.client_identifier <p>The name that the client wants the DHCP server to use to identify it. Enter the name as a quoted string. For example: "phantastica".</p>	NULL char *
RFC2131 Initialization Delay identifier DHCPC_RFC2131_INIT_DELAY ipdhcpc.rfc2131_init_delay <p>Either "1" or "0". If set to "1", the DHCP client waits a random number of seconds before resending DHCPDISCOVER messages to locate DHCP servers. If set to "0" there is no delay.</p>	"1" char *

Table 6-2 DHCP-Client Build Parameters (cont'd)

Workbench Description, Parameter Name, and sysvar	Default Value & Data Type
RFC2131 Exponential Back-off Delay DHCPC_RFC2131_EXP_BACKOFF ipdhcpc.rfc2131_exponential_backoff <p>Either "1" or "0". If set to "1", the DHCP client waits a random number of seconds before resending DHCPREQUEST messages. If set to "0" there is no delay.</p>	<p>"1"</p> <p>char *</p>
Number of DHCP client retries DHCPC_DISCOVER_RETRIES ipdhcpc.discover_retries <p>If the DHCP client sends a DHCPDISCOVER message to find a DHCP server and does not get a response, this parameter determines the maximum number of times the client retries.</p>	<p>"4"</p> <p>char *</p>
DHCP offer time-out in milliseconds DHCPC_OFFER_TIMEOUT ipdhcpc.offer_timeout <p>The length of time, in milliseconds, that the DHCP client waits for a response to a DHCPDISCOVER message before it resends the message.</p>	<p>"2000"</p> <p>char *</p>

Table 6-2 DHCP-Client Build Parameters (cont'd)

Workbench Description, Parameter Name, and sysvar	Default Value & Data Type
Interface specific list of requested dhcp options DHCP_IF_REQ_OPTS_LIST ipdhcpc.interfaceName.requested_options	"" char *
The DHCP options that the client wants the server to provide to an interface or interfaces.	
Use the following format for the configuration parameter:	
<i>interfaceName=options ; interfaceName=options ; interfaceName=options ; ...</i>	
where <i>interfaceName</i> is the name of an interface and <i>options</i> is a comma separated list of DHCP options (see RFC 2132). For example:	
<code>"eth0=1,6,28,42;eth1=1,3,5,40,41"</code>	
For the sysvar, use the following format:	
<code>sysvar set ipdhcpc.interfaceName.requested_options options</code>	
For example:	
<code>sysvar set ipdhcpc.eth0.requested_options 1,6,28,42</code>	

Table 6-2 DHCP-Client Build Parameters (cont'd)

Workbench Description, Parameter Name, and sysvar	Default Value & Data Type
Interface specific list of client identifier	""
DHCPC_IF_CLIENT_ID_LIST	
ipdhcpc.interfaceName.client_identifier	char *
<p>The name or names the client wants the DHCP server to use to identify it on an interface or interfaces.</p> <p>Use the following format for the configuration parameter:</p> <pre>interfaceName=clientID; interfaceName=clientID; interfaceName=clientID; ...</pre> <p>where <i>interfaceName</i> is the name of an interface and <i>clientID</i> is a character string identifying the client on an interface. For example:</p> <pre>"eth0=id1;eth1=id2;eth2=id3"</pre> <p>For the sysvar, use the following format:</p> <pre>sysvar set ipdhcpc.interfaceName.client_identifier clientID</pre> <p>For example:</p> <pre>sysvar set ipdhcpc.eth0.client_identifier id1</pre>	

Table 6-2 **DHCP-Client Build Parameters** (cont'd)

Workbench Description, Parameter Name, and sysvar	Default Value & Data Type
Install dhcp client callback routine DHCPC_INSTALL_CALLBACK_HOOK sysvar: None. If set to TRUE , you can implement the ipdhcpc_option_callback() routine and include it in the build. If set to FALSE , the default version of the callback, which does not perform any operations, is called. For information on ipdhcpc_option_callback() , (see 6.4 Implementing the ipdhcpc_option_callback() Routine , p.101.	FALSE BOOL
Option callback routine DHCPC_OPTION_CALLBACK_HOOK sysvar: None. If the Install dhcp client callback routine (DHCPC_INSTALL_CALLBACK_HOOK) parameter is set to TRUE , you must set this parameter to point to your implementation of the ipdhcpc_option_callback() routine. With the ipdhcpc_option_callback() routine, you can define your own operations on individual DHCP options (see 6.4 Implementing the ipdhcpc_option_callback() Routine , p.101.	NULL funcptr

6.3 Using Shell Commands

There are no shell commands specific to the DHCP client. You can use the **ipd** shell command to start and stop the DHCP-client daemon. You can use the **ifconfig** shell command to enable or disable DHCP on an interface and to display the current status of an interface. The various **ipd** and **ifconfig** commands are as follows:

```
ipd start ipdhcpc
    Start the DHCP client (after stopping it with ipd kill ipdhcpc).
```

```
ipd kill ipdhcpc
    Stop the DHCP client.

ifconfig interfaceName dhcp
    Enable the interface interfaceName for DHCP and start a DHCP session on it.

ifconfig interfaceName -dhcp
    Close the current session on interface interfaceName and disable DHCP on it.

ifconfig interfaceName
    Show the current status and configuration of interface interfaceName. For
    example:

    -> ifconfig "eth0"

eth0 Link type:6 HWaddr 00:0a:01:02:44:02
inet 10.1.1.231 broadcast 10.1.255.255 mask 255.255.0.0
inet6 unicast FEC0::1:20A:1FF:FE02:4402
inet6 unicast FEC0::A9:20A:1FF:FE02:4402
inet6 unicast FEC0::1:0:0:0:82
inet6 multicast FF02::1%eth0
inet6 unicast FE80::20A:1FF:FE02:4402%eth0
UP RUNNING BROADCAST MULTICAST DHCP
MTU:1500 metric:1 rtab:0
RX packets:62 mcast:1 errors:0 dropped:2
TX packets:11 mcast:3 errors:0
collisions:0 unsupported proto:45
RX bytes:4989 TX bytes:1562
```

6.4 Implementing the `ipdhcpc_option_callback()` Routine

There are many DHCP options that the DHCP client does not handle directly (see [6.4.1 DHCP Options Not Initially Implemented in the Client](#), p. 103, [Table 6-3](#)). You can implement your own handler for these options through the `ipdhcpc_option_callback()` routine. You can also implement this routine to provide special treatment of any of the DHCP options that the client already handles. For a list of those options, see [6.1.1 Conformance to Standards](#), p. 94, [Table 6-1](#).

For each option in a DHCP-server message, the client calls `ipdhcpc_option_callback()` before it performs any other action involving the option. The signature of `ipdhcpc_option_callback()` is:

```
IP_PUBLIC Ip_err ipdhcpc_option_callback (Ip_u8 * option)
```

The **option** parameter points to an option in a DHCP-server message, where:

- **option[0]** is a DHCP option code.
- **option[1]** is the length of **option**, in bytes.
- **option[2]** is the first element containing data for the option.

Write your **ipdhcpc_option_callback()** routine to return one of the following two values:

IP_TRUE

Indicates that the DHCP client should continue to process the option.

IP_FALSE

Indicates that the DHCP client should do nothing further with the option.

If you do not implement the **ipdhcpc_option_callback()** routine, the client uses a default version of the routine that performs no operations on any options and returns **IP_TRUE**.

Example 6-1 **A Sample Implementation of ipdhcpc_option_callback()**

The following sample implementation of the **ipdhcpc_option_callback()** routine checks the **option** parameter to see if it contains the DHCP option code for specifying a host name—**IPDHCP_OPTCODE_HOST_NAME**. If it does, the routine assigns the host name specified in the **option** parameter to the client.

```
/*
 * DHCP_INSTALL_CALLBACK_HOOK = TRUE
 * DHCP_OPTION_CALLBACK_HOOK = myDhcpClientHook
 */

#include <ipcom_type.h>
#include <ipdhcpc_config.h>
#include <ipdhcpc.h>

Ip_err myDhcpClientHook (Ip_u8 * option)
{
    char myHostName[20];
    unsigned char length;

    switch (*option)
    {
        case IPDHCP_OPTCODE_HOST_NAME:
            gethostname (myHostName, 20);
            printf ("Host name before change is %s\n", myHostName);
            option++;
            length = *option;
            option++;
            strcpy (myHostName, (char *)option);
            myHostName[length] = '\0'; /* Insert \0 at the end */
    }
```

```
        sethostname (myHostName, length);
        gethostname (myHostName, 20);
        printf ("Host name changed to %s\n", myHostName);
        break;
default:
    printf ("Got option %d. Do nothing with it\n", *option);
    break;
}
}
```

6.4.1 DHCP Options Not Initially Implemented in the Client

The table that follows lists macro names of DHCP option codes that are not implemented in the Wind River DHCP client, but that you can implement through the `ipdhcpc_option_callback()` routine. DHCP options that have built-in implementations in the client are listed in [6.1.1 Conformance to Standards](#), p.94, [Table 6-1](#).

Table 6-3 **Macros for DHCP Options Not Implemented in the Client**

Macro for DHCP Option Code	Code	Option Value Type
IPDHCPC_OPTCODE_ALL_SUBNETS_LOCAL	27	boolean
IPDHCPC_OPTCODE_ARP_CACHE_TIMEOUT	35	32-bits
IPDHCPC_OPTCODE_BOOT_SIZE	13	16-bits
IPDHCPC_OPTCODE_BROADCAST_ADDRESS	28	IPv4 address
IPDHCPC_OPTCODE_CLIENT_IDENTIFIER	61	(varies)
IPDHCPC_OPTCODE_COOKIE_SERVERS	8	list of IPv4 addresses
IPDHCPC_OPTCODE_DEFAULT_IP_TTL	23	8-bits
IPDHCPC_OPTCODE_DEFAULT_TCP_TTL	37	8-bits
IPDHCPC_OPTCODE_DHCP_OPTION_OVERLOAD	52	8-bits
IPDHCPC_OPTCODE_DHCP_REQUESTED_ADDRESS	50	IPv4 address
IPDHCPC_OPTCODE_EXTENSIONS_PATH	18	string
IPDHCPC_OPTCODE_FINGER_SERVERS	73	list of IPv4 addresses
IPDHCPC_OPTCODE_FONT_SERVERS	48	list of IPv4 addresses

Table 6-3 **Macros for DHCP Options Not Implemented in the Client** (cont'd)

Macro for DHCP Option Code	Code	Option Value Type
IPDHCPC_OPTCODE_HOME_AGENTS	68	list of IPv4 addresses
IPDHCPC_OPTCODE_HOST_NAME	12	string
IPDHCPC_OPTCODE_IEEE802_3_ENCAPSULATION	36	boolean
IPDHCPC_OPTCODE_IMPRESS_SERVERS	10	list of IPv4 addresses
IPDHCPC_OPTCODE_INTERFACE_MTU	26	16-bits
IPDHCPC_OPTCODE_IP_FORWARDING	19	boolean
IPDHCPC_OPTCODE_IRC_SERVERS	74	list of IPv4 addresses
IPDHCPC_OPTCODE_LOG_SERVERS	7	list of IPv4 addresses
IPDHCPC_OPTCODE_LPR_SERVERS	9	list of IPv4 addresses
IPDHCPC_OPTCODE_MASK_SUPPLIER	30	boolean
IPDHCPC_OPTCODE_MAX_DGRAM_REASSEMBLY	22	16-bits
IPDHCPC_OPTCODE_MERIT_DUMP	14	string
IPDHCPC_OPTCODE_NAME_SERVERS	5	list of IPv4 addresses
IPDHCPC_OPTCODE_NETBIOS_DD_SERVER	45	list of IPv4 addresses
IPDHCPC_OPTCODE_NETBIOS_NAME_SERVERS	44	list of IPv4 addresses
IPDHCPC_OPTCODE_NETBIOS_NODE_TYPE	46	8-bits
IPDHCPC_OPTCODE_NETBIOS_SCOPE	47	string
IPDHCPC_OPTCODE_NIS_DOMAIN	40	string
IPDHCPC_OPTCODE_NIS_SERVERS	41	list of IPv4 addresses
IPDHCPC_OPTCODE_NON_LOCAL_SOURCE_ROUTING	20	boolean
IPDHCPC_OPTCODE_PATH_MTU_AGING_TIMEOUT	24	32-bits
IPDHCPC_OPTCODE_PATH_MTU_PLATEAU_TABLE	25	list of 16-bit values
IPDHCPC_OPTCODE_PERFORM_MASK_DISCOVERY	29	boolean

Table 6-3 **Macros for DHCP Options Not Implemented in the Client** (cont'd)

Macro for DHCP Option Code	Code	Option Value Type
IPDHCPC_OPTCODE_POLICY_FILTER	21	list of IPv4 address pairs
IPDHCPC_OPTCODE_RESOURCE_LOCATION_SERVERS	11	list of IPv4 addresses
IPDHCPC_OPTCODE_ROOT_PATH	17	string
IPDHCPC_OPTCODE_ROUTER_DISCOVERY	31	boolean
IPDHCPC_OPTCODE_ROUTER_SOLICITATION_ADDRESS	32	IPv4 address
IPDHCPC_OPTCODE_STATIC_ROUTES	33	list of IPv4 address pairs
IPDHCPC_OPTCODE_SWAP_SERVER	16	IPv4 address
IPDHCPC_OPTCODE_TCP_KEEPALIVE_GARBAGE	39	boolean
IPDHCPC_OPTCODE_TCP_KEEPALIVE_INTERVAL	38	32-bits
IPDHCPC_OPTCODE_TIME_OFFSET	2	32-bits
IPDHCPC_OPTCODE_TIME_SERVERS	4	list of IPv4 addresses
IPDHCPC_OPTCODE_TRAILER_ENCAPSULATION	34	boolean
IPDHCPC_OPTCODE_VENDOR_CLASS_IDENTIFIER	60	32-bit unsigned integer
IPDHCPC_OPTCODE_VENDOR_ENCAPSULATED_OPTIONS	43	string
IPDHCPC_OPTCODE_X_DISPLAY_MANAGER	49	list of IPv4 addresses

7

Wind River DHCPv6: Server and Relay Agent

7.1 Introduction 107

7.2 Assigning Client-specific Authentication Keys 110

7.1 Introduction

This chapter describes the Wind River DHCPv6 server and relay agent implementations. For a general overview of DHCP, see [3. Wind River DHCP and DHCPv6: Overview](#).

The Wind River DHCPv6 *server* provides IP addresses and option settings to DHCPv6 clients.

A *relay agent* is an Internet host or router that passes DHCP messages between DHCPv6 clients and DHCPv6 servers. There can be multiple relay agents between a client and a server.

Including the DHCPv6 Server or Relay Agent in a Build

To include the DHCPv6 server or relay agent in a VxWorks build, include the **DHCP6 Server** (INCLUDE_IPDHCP6) build component. In addition, to have access to shell commands for the server or relay agent, include the **IPCOM DHCP6 Server commands** (INCLUDE_IPDHCP6_CMD) build component.

Build Configuration Parameters

The **DHCP6 Server** (`INCLUDE_IPDHCP6`) build component has several configuration parameters. [Table 7-1](#) lists and describes each parameter. See [3.1.3 Build Configuration Parameters and sysvars](#), p.46 for more information on how to set these parameters.

Table 7-1 **DHCP Server Build Configuration Parameters**

Workbench Description, Parameter Name, and sysvar	Default Value & Data Type
DHCPV6 Authentication Realm <code>DHCP6_AUTHENTICATION_REALM</code> <code>ipdhcps6.authrealm</code> The Authentication Realm as described in section 21.4.1 of RFC 3315. This parameter, in addition to setting the DHCPV6 authentication realm, also controls whether the DHCPv6 server will generate DHCPv6 replies using authentication at all. If you leave this parameter in its default value of "" the server will not do authenticated exchanges. Set this parameter to anything other than "" to turn on the authentication capability.	"" char *
DCHPV6 Authentication HMAC-MD5 Key <code>DHCP6_HMAC_MD5_SECRET</code> <code>ipdhcps6.authkey</code> The default authentication key that the server will use for message authentication, if a client requests authentication but the server does not know of a client-specific key for that client. (To set a client-specific authentication key for a client, see 7.2 Assigning Client-specific Authentication Keys , p.110.) Specify the key as an ASCII string, starting with the prefix "0x" and followed by an even number of hexadecimal digits, for example: "0x0123456789abcdef". The string length that you can give in the kernel configurator is limited to 130 bytes, which translates to a key length of up to 64 bytes. If you leave this parameter at its default value of "", only clients that have had a specific key assigned to them will be able to use authenticated DHCPv6 communications.	"" char *

Table 7-1 DHCP Server Build Configuration Parameters (cont'd)

Workbench Description, Parameter Name, and sysvar	Default Value & Data Type
Mode DHCPv6_MODE ipdhcps6.mode Determines whether the router runs as a DHCPv6 server ("server") or relay agent ("relay").	"server" char *
Interface Name DHCPv6_DUID_IFNAME ipdhcps6.duid.ifname The name of the interface whose link-layer address the server is to use as its DHCPv6 unique ID (DUID), for example: "eth0". You must enter a value for this parameter or the DHCP server will not start. When the server starts, the DUID is written to the system log.	"" char *
Allow Rapid Commit DHCPv6_ALLOW_RAPID_COMMIT ipdhcps6.server.allow_rapid_commit If set to "yes", the server allows rapid commits.	"yes" char *

Table 7-1 DHCP Server Build Configuration Parameters (cont'd)

Workbench Description, Parameter Name, and sysvar	Default Value & Data Type
Hop Count Limit DHCPs6_MAXHOP_COUNT ipdhcps6.relay.hop_count_limit	"32" char *
(Relay agents only) The maximum number of hops allowed when forwarding DHCPv6 requests.	
Interface Relay Map List DHCPs6_IF_RELAY_MAP_LIST ipdhcps6.relay.map <i>inInterfaceName</i>	"" char *
(Relay agents only) For each interface that receives DHCP messages, this parameter specifies either an outgoing interface to use for multicasting messages or a unicast destination address.	
The format for a configuration parameter entry is:	
<i>"inInterfaceName=[outInterfaceName destinationAddress]"</i>	
Enter interface pairs as a string, with individual entries separated by semicolons. The following are configuration parameter examples:	
<pre>"eth0=eth1" "eth0=eth1;eth2=2001:DB8:123::2:1"</pre>	
The following are the equivalent sysvar examples:	
<pre>sysvar set ipdhcps6.relay.map.eth0 eth1 sysvar set ipdhcps6.relay.map.eth0 eth1 sysvar set ipdhcps6.relay.map.eth2 2001:DB8:123::2:1</pre>	

7.2 Assigning Client-specific Authentication Keys

To assign a separate authentication key for a specific clients, based on their client DUID, use the following routines:

ipdhcps6_authdb_create_ll_duid()

```
/* create a DUID based on link-layer address: section 9.4 RFC3315*/  
IP_GLOBAL Ipdhcp6_duid *  
    ipdhcps6_authdb_create_ll_duid (int len, Ip_u16 hwtype, Ip_u8 * macaddr)
```

The parameters to this routine are as follows:

len – length of the hardware address (mac address)

hwtype – the hardware type code. Most likely to be 0x1 (ethernet)

macaddr – pointer to a byte string containing the hardware address.



NOTE: Currently, **len** is ignored. **ipdhcps6_authdb_create_ll_duid()** uses only the first six bytes of **macaddr** to identify the client.

This routine returns a pointer to the newly created DUID structure, or **IP_NULL** if the routine fails.

ipdhcps6_authdb_add_client_authkey()

```
/* associate an authentication key with a specific client */  
IP_GLOBAL IP_err  
    ipdhcps6_authdb_add_client_authkey (Ipdhcp6_duid * client_duid,  
                                         Ip_u8 * key, int keylen)
```

Given a previously created client DUID structure, and a key/key length pair, this routine creates an entry in the internal authentication database assigning the key to the given client.

This routine returns **IPCOM_SUCCESS** if successful, or **IPCOM_ERR_FAILED** if not.

Example 7-1 Example Code: Assigning an Authentication Key to a Client

The client has an ethernet mac address of 01:02:03:04:05:06, and we wish to assign a specific authentication key of **"this is a secret"**.

```
void assign_example (void)  
{  
    Ipdhcp6_duid * client_duid;  
  
    Ip_u8 macaddr[] = { 0x1, 0x2, 0x3, 0x4, 0x5, 0x6 };  
    char * key      = "this is a secret";  
    int   keylen    = strlen (key);  
  
    client_duid = ipdhcps6_authdb_create_ll_duid (6, 0x1, macaddr);  
    if (IP_NULL == client_duid)  
    {  
        LOG_ERROR_SOMEHOW ("couldnt create client duid");  
        return;  
    }  
}
```

```
if (IPCOM_SUCCESS !=
    ipdhcps6_authdb_add_client_authkey (client_duid, (IP_u8 *) key, keylen))
{
    LOG_ERROR_SOMEHOW ("could not associate key with client duid");
}
```

The return value from **ipdhcps6_authdb_create_ll_duid()** is a pointer to a memory area obtained by a call to **ipcom_malloc()**. Once you have called **ipdhcps6_authdb_add_client_authkey()** to associate the key with the client DUID in that memory region, you may release this memory area by calling **ipcom_free()**. However, you may wish to keep this DUID (and the memory region that contains it) for a later call to **ipdhcps6_authdb_delete_client_authkey()**.



NOTE: This assignment procedure is sufficient for clients that send DUID's of type LL (link-layer-address) or LLT (link-layer-address + time). Assigning a client-specific key to clients which send an EN (enterprise number) type DUID is currently unsupported (but is planned in a future release).

ipdhcps6_authdb_delete_client_authkey()

```
/* To delete a specific clients auth key entry */
IP_GLOBAL IP_err
    ipdhcps6_authdb_delete_client_authkey (Ipdhcps6_duid *client_duid);
```

This call deletes any client-specific key associated with the given client DUID. Returns **IPCOM_SUCCESS** if successful, **IPCOM_ERR_FAILED** on failure.

8

Wind River DHCPv6: Client

- 8.1 Introduction 113
- 8.2 Including the DHCPv6 Client in a Build 120
- 8.3 Using Shell Commands 132

8.1 Introduction

This chapter describes the Wind River DHCPv6 client. For a general overview of DHCP and DHCPv6, see [3. *Wind River DHCP and DHCPv6: Overview*](#).

Client Overview

The Wind River DHCP client runs as a single process that listens to both ICMPv6 router advertisements and routing socket messages. The client sends queries to DHCP servers to ask for either stateless or stateful information, depending on how you configure the client. In the case of stateful information, the client periodically communicates with the server to update configuration information and extend any existing leases.

8.1.1 Configuring the DHCPv6 Client

You can statically configure a number of client parameters. Most of the static parameter settings either determine the kind of information that interfaces can receive or give hints to DHCPv6 servers about the settings the client prefers (see [Build Configuration Parameters](#), p.120). To display information about configuration settings and to refresh or release the settings for interfaces, you can use shell commands (see [8.3 Using Shell Commands](#), p.132).

Some of the static configuration parameters allow you to specify whether or not an interface handles specific types of configuration information. A third option is to specify that an interface operates in *automatic* mode, as described in the next section.

Automatic Mode

Router advertisement messages contain an **M** field, for a **Managed address configuration** flag, and an **O** field, for an **Other stateful configuration** flag (see RFC 2461: *Neighbor Discovery for IP Version 6*). When you set a static build parameter to **automatic**, the affected interfaces operate in accord with the **M** and **O** settings in the most recently received router advertisement message. The settings are interpreted as follows:

- If the **Managed address configuration** flag (**M**) is set, interfaces configure IPv6 addresses using stateful autoconfiguration.
- If the **Other stateful configuration** flag (**O**) is set, interfaces use stateful autoconfiguration to obtain information other than IPv6 addresses.

If the **Managed address configuration** flag is set, the **Other stateful configuration** flag must also be set. The client cannot use stateful address autoconfiguration without accepting additional stateful information from a DHCP server.

Assigning Server-specific Authentication Keys

To assign an authentication key to a specific DHCPv6 server, based on its server DUID, use the following routines:

ipdhcpc6_authdb_create_ll_duid()

```
/* create a DUID based on link-layer address: section 9.4 RFC3315 */  
Ipdhcpc6_duid *  
    ipdhcpc6_authdb_create_ll_duid (int len, Ip_u16 hwtype, Ip_u8 * macaddr)
```

The parameters to this routine are as follows:

len – the length of the hardware address (mac address)

hwtype – the hardware type code, usually 0x1 (ethernet)

macaddr – a pointer to a byte string containing the hardware address



NOTE: The **len** parameter is currently ignored;
ipdhcpc6_authdb_create_ll_duid() uses only the first six bytes of **macaddr** to identify the server.

ipdhcpc6_authdb_create_ll_duid() returns a pointer to the newly created DUID structure, or **IP_NULL** if the routine fails.

ipdhcps6_authdb_add_server_authkey()

```
/* associate an authentication key with a specific server */
Ip_err ipdhcps6_authdb_add_server_authkey (Ipdhcpc6_duid * server_duid,
                                           Ip_u8 * key, int keylen)
```

Given a previously created server DUID structure, and a key/key length pair, this routine creates an entry in the internal authentication database assigning the key to the given server.

This routine returns **IPCOM_SUCCESS** if successful, or **IPCOM_ERR_FAILED** if not.

Example 8-1 **Assigning an Authentication Key to a Server**

In this example, the server has an ethernet mac address of 01:02:03:04:05:06, and we want to assign a specific authentication key of "this is a secret".

```
void assign_example (void)
{
    Ipdhcpc6_duid * server_duid;

    Ip_u8 macaddr[] = { 0x1, 0x2, 0x3, 0x4, 0x5, 0x6 };
    char * key      = "this is a secret";
    int   keylen    = strlen (key);

    server_duid = ipdhcpc6_authdb_create_ll_duid (6, 0x1, macaddr);
    if (IP_NULL == client_duid)
    {
        LOG_ERROR_SOMEHOW ("could not create server duid");
        return;
    }

    if (IPCOM_SUCCESS !=
        ipdhcpc6_authdb_add_server_authkey (server_duid, (Ip_u8 *) key, keylen))
    {
        LOG_ERROR_SOMEHOW ("couldnt associate key with server duid");
    }
}
```

The return value from **ipdhcpc6_authdb_create_ll_duid()** is a pointer to a memory area obtained with a call to **ipcom_malloc()**. After you call **ipdhcpc6_authdb_add_server_authkey()** to associate the key with the server DUID, you may release the memory region containing the DUID by calling **ipcom_free()**. However, you may wish to keep it around for a later call to **ipdhcpc6_authdb_server_no_auth()**.



NOTE: This assignment procedure is sufficient for servers that send DUIDs of type LL (link-layer-address) or LLT (link-layer-address + time). Assigning a server-specific key to servers that send an EN (enterprise number) type DUID is currently not supported.

ipdhcpc6_authdb_server_no_auth()

```
/* To delete a specific servers auth key entry */
Ip_err ipdhcpc6_authdb_server_no_auth (Ipdhcpc6_duid * client_duid);
```

This call deletes any server-specific key associated with the given server DUID, then returns **IPCOM_SUCCESS** if successful, or **IPCOM_ERR_FAILED** otherwise.

Run-time Configuration of the DHCPv6 Authentication Parameters

A DHCPV6-enabled device will probably need to configure itself in the field, after manufacture. If you set the **DHCPC6_AUTHENTICATION_REALM** configuration parameter to anything other than the default setting (""), the DHCPV6 client task will block on startup, waiting for this run-time configuration to take place. This blocking behavior does not occur if you set **DHCPC6_AUTHENTICATION_REALM** to the default ("").

After a device completes such configuration, it must call **ipdhcpc6_user_authdb_config_finished()** to notify the DHCPV6 client that configuration is complete. Only then will the DHCPV6 client become operational.

Write a routine to do this run-time configuration, and invoke this routine at device startup, for instance, from **usrAppInit()**.

The simplest run-time configuration routine (which does nothing except to notify the DHCPV6 client that it can go ahead and start up) would be the following:

```
extern void ipdhcpc6_user_authdb_config_finished (void);
runtimeConfigureDHCPV6auth (void)
{
    ipdhcpc6_user_authdb_config_finished ();
}
```

A device that needs to configure authentication parameters, but does not have any information about its configuration available at kernel configuration time, must set

the `DHCPC6_AUTHENTICATION_REALM` kernel configuration parameter to some non-empty-string value, such as "yes", and then provide a runtime configuration routine that configures the device at startup using values obtained from the end user, NVRAM storage, and so forth.

Call the following routines to configure the authentication parameters:

```
Ip_err ipdhcpc6_set_auth_realm (int len, Ip_u8 * realm);
```

Set the authentication realm for the DHCPV6 client. This overwrites the `DHCPC6_AUTHENTICATION_REALM` configuration parameter.

Parameters:

len – the length of the byte string pointed to by the **realm** parameter.

realm – a pointer to the realm string. It does not need to be null-terminated.

This routine returns `IPCOM_SUCCESS` if successful, or `IPCOM_ERR_FAILED` if the supplied realm length is too large.

The `DHCPC6_AUTHENTICATION_REALM` parameter is limited to 128 bytes.

```
Ip_err ipdhcpc6_set_default_auth_key (int len, Ip_u8 * key);
```

Set the default authentication key for the DHCPV6 client. This overwrites the `DHCPC6_HMAC_MD5_SECRET` configuration parameter.

Parameters:

len – the length of the byte string pointed to by the **key** parameter. The key length is limited to 64 bytes.

key – a pointer to the authentication key, supplied as a raw byte sequence. It may contain null bytes.

This routine returns `IPCOM_SUCCESS` if successful, or `IPCOM_ERR_FAILED` if the supplied key is too large.

```
Ipdhcpc6_duid * ipdhcpc6_authdb_create_ll_duid  
(int len, Ip_u16 hwtype, Ip_u8 * macaddr)
```

```
Ip_err ipdhcpc6_authdb_add_server_authkey  
(Ipdhcpc6_duid * server_duid, Ip_u8 * key, int keylen)
```

These two routines are described in [Assigning Server-specific Authentication Keys](#), p.114.

```
void ipdhcpc6_user_authdb_config_finished (void);
```

Call this routine to notify the DHCPV6 client that initial run-time configuration is complete, and so the DHCPV6 client may start.

Initializing the client replay-detection counters

DHCPv6 authentication incorporates an anti-replay mechanism to protect against replay attacks. Each client/server pair maintains an 8-byte counter which must monotonically increase with each DHCPv6 message sent. This can be problematic if a client has previously established a session with a given server, and then experiences a reboot. Unless the client has some way of initializing the starting value of this counter to a value greater than its value before it rebooted, it may be refused authenticated service with that server.

Wind River recommends that you tie the replay counter to the time in some fashion. Some DHCPv6 clients and servers use an NTP format timestamp as their anti-replay counter. If your target environment has a time/date reference available to it, or a number-of-reboots counter, or something of this sort, Wind River recommends that you use that information at start up to initialize the anti-replay counter. Use the following routine to set the anti-replay counter:

```
IP_GLOBAL void ipdhcpc6_set_replay_counters (Ip_u32 high, Ip_u32 low);
```

- **high** –upper 4 bytes of the anti-replay counter
- **low** – lower 4 bytes of the anti-replay counter

You can obtain the current value of the replay counter at any time by calling the following routine:

```
IP_GLOBAL void ipdhcpc6_get_replay_counters (Ip_u32 *high, Ip_u32 *low);
```

- **high** – pointer to location where upper 4 bytes of the anti-replay counter will be written
- **low** – pointer to location where lower 4 bytes of the anti-replay counter will be written

8.1.2 Conformance to Standards

RFC 3315: *Dynamic Host Configuration Protocol for IPv6*, is the primary specification of DHCP for IPv6, although a number of other RFCs are also relevant to the implementation of a DHCPv6 client.

Implementation of RFC 3315, Dynamic Host Configuration Protocol for IPv6

The Wind River DHCPv6 client implements the client portions of RFC 3315, with the following exceptions:

- **Reconfigure messages**

The Wind River DHCPv6 client does not include Reconfigure messages. Reconfigure messages are described in RFC 3315, section 5.3, “DHCP Message Types.” This means the Reconfiguration options **OPTION_RECONF_MSG** and **OPTION_RECONF_ACCEPT** do not work.

- **Vendor or user-defined options**

The Wind River DHCPv6 client does not allow vendor- or user-defined configuration options. This means the following options do not work: **OPTION_USER_CLASS**, **OPTION_VENDOR_CLASS**, and **OPTION_VENDOR_OPTS**.

- **Temporary address assignments**

The Wind River DHCPv6 client does not include the assignment of temporary addresses, as described in RFC 3041: *Privacy Extensions for Stateless Address Autoconfiguration in IPv6*.

DHCPv6 Configuration Options Implemented by the Wind River DHCPv6 Client

The Wind River DHCPv6 client implements all configuration options in the following RFCs:

- RFC 3646: *DNS Configuration options for Dynamic Host Configuration Protocol for IPv6*
- RFC 4075: *Simple Network Time Protocol (SNTP) Configuration Option for DHCPv6*
- RFC 4242: *Information Refresh Time Option for Dynamic Host Configuration Protocol for IPv6*

The Wind River DHCPv6 client does not include vendor-specific and user-defined configuration options or configuration options contained in RFCs that are not listed here.

Stateless DHCP Configuration

The DHCPv6 client implements all client-relevant features of RFC 3736: *Stateless Dynamic Host Configuration Protocol (DHCP) Service for IPv6*.

8.2 Including the DHCPv6 Client in a Build

To include the DHCPv6 client in a VxWorks build, include the **DHCP6 Client** (`INCLUDE_IPDHCP6`) build component.

Build Configuration Parameters

The **DHCP6 Client** (`INCLUDE_IPDHCP6`) build component provides a number of configuration parameters. [Table 8-1](#) lists and describes each parameter. See [3.1.3 Build Configuration Parameters and sysvars](#), p.46 for more information on setting these parameters.

Table 8-1 DHCPv6 Client Build Parameters

Workbench Description, Parameter Name, and sysvar	Default Value & Data Type
<p>Workbench description: none</p> <p>DHCPv6_AUTHENTICATION_REALM</p> <p>sysvar: none</p> <p>The Authentication Realm as described in section 21.4.1 of RFC 3315.</p> <p>This parameter, in addition to setting the DHCPv6 authentication realm, also controls whether the DHCPv6 client will generate DHCPv6 requests asking for authenticated DHCPv6 service. If you leave this parameter in its default value of "" the server will not do authenticated exchanges. Set this parameter to anything other than "" to turn on the authentication capability.</p> <p>The authentication realm is size limited to 128 bytes.</p> <p>See Run-time Configuration of the DHCPv6 Authentication Parameters, p.116 for more details.</p>	<p>""</p> <p>char *</p>
<p>Workbench description: none</p> <p>DHCPv6_HMAC_MD5_SECRET</p> <p>sysvar: none</p> <p>The default authentication key that the client will use for message authentication, if you configure the client for authenticated DHCPv6 operation and it receives a message from a server for which you have not configured a specific key. (To learn how to assign a key to a server, see Assigning Server-specific Authentication Keys, p.114.)</p> <p>Specify the key as an ASCII string, starting with the prefix "0x" and followed by an even number of hexadecimal digits, for example: "0x0123456789abcdef". The string length that you can give in the kernel configurator is limited to 130 bytes, which translates to a key length of up to 64 bytes.</p> <p>If you leave this parameter at its default value of "", the client will use only those replies from servers to which the client has a specific key mapped for DHCPv6 exchanges.</p> <p>See Run-time Configuration of the DHCPv6 Authentication Parameters, p.116 for more details.</p>	<p>""</p> <p>char *</p>

Table 8-1 **DHCPv6 Client Build Parameters** (cont'd)

Workbench Description, Parameter Name, and sysvar	Default Value & Data Type
DUID Type DHCPC6_DUID_TYPE ipdhcpc6.duid.type The type of DHCP Unique ID (DUID) the client uses. The possible values are: <ul style="list-style-type: none">▪ "ll" (link-layer)▪ "llt" (link-layer plus timestamp)▪ "en" (enterprise)	"ll" char *
DUID Interface DHCPC6_DUID_IF ipdhcpc6.duid.if For ll and llt DUIDs, the interface whose link-layer address the client is to use, for example: "eth0". If you do not specify an interface, the client selects the first DHCP-enabled interface it finds.	"" char *
DUID EN Number DHCPC6_DUID_EN_NUM ipdhcpc6.duid.en.number For en DUIDs, the enterprise number to use when the client generates a DUID. If you set this value, you must make sure that no value is set for the DUID EN Value (DHCPC6_DUID_EN_VAL) parameter (see the next table entry).	"10000" char *
DUID EN Value DHCPC6_DUID_EN_VAL ipdhcpc6.duid.value For en DUIDs, the enterprise value to use when the client generates a DUID. If you set this value, you must make sure that no value is set for the DUID EN Number (DHCPC6_DUID_EN_NUM) parameter (see the preceding table entry). The value entered for this parameter must be a string, but the string can contain a hexadecimal value. Each of the following are valid entries: "0xabcdef" "duid123"	"0xabcdef" char *

Table 8-1 DHCPv6 Client Build Parameters (cont'd)

Workbench Description, Parameter Name, and sysvar	Default Value & Data Type
Interface Status List DHCPC6_IF_ENUM_LIST ipdhcpc6.if.enum.interfaceName="status"	"" char *
<p>Whether or not individual interfaces are enabled for DHCP. For each interface, enter one of the following values: "enable", "disable", or "automatic". For the interpretation of "automatic", see Automatic Mode, p.114.</p> <p>The following are configuration parameter examples:</p> <pre>"eth0=automatic" "eth0=automatic;eth1=enable"</pre> <p>The following are the equivalent sysvar examples:</p> <pre>sysvar set ipdhcpc6.enum.eth0 automatic sysvar set ipdhcpc6.enum.eth0 automatic sysvar set ipdhcpc6.enum.eth1 enable</pre>	
Interface Rapid Commit Status List DHCPC6_IF_RAPID_COMMIT_LIST ipdhcpc6.if.interfaceName.rapid_commit="status"	"" char *
<p>Determines whether individual interfaces can use the DHCPv6 Rapid Commit option. For each interface, enter either "enable" or "disable".</p> <p>The following are configuration parameter examples:</p> <pre>"eth0=enable" "eth0=enable;eth1=enable"</pre> <p>The following are the equivalent sysvar examples:</p> <pre>sysvar set ipdhcpc6.if.eth0.rapid_commit enable sysvar set ipdhcpc6.if.eth0.rapid_commit enable sysvar set ipdhcpc6.if.eth1.rapid_commit enable</pre>	

Table 8-1 **DHCPv6 Client Build Parameters** (cont'd)

Workbench Description, Parameter Name, and sysvar	Default Value & Data Type
Interface Information Only Status List DHCPC6_IF_INFORMATION_ONLY_LIST ipdhcpc6.if.interfaceName.information_only="status" Whether or not an interface only requests stateless information. For each interface, enter either "enable" or "disable" . The following are configuration parameter examples: "eth0=enable" "eth0=enable;eth1=enable" The following are the equivalent sysvar examples: sysvar set ipdhcpc6.if.eth0.information_only enable sysvar set ipdhcpc6.if.eth0.information_only enable sysvar set ipdhcpc6.if.eth1.information_only enable	"" char *
Interface DNS Status List DHCPC6_IF_DNS_LIST ipdhcpc6.if.interfaceName.dns="status" For interfaces that are able to receive stateless information, whether or not an individual interface can request information about DNS servers and Domain Search Lists. For each interface, enter either "enable" or "disable" . The following are configuration parameter examples: "eth0=enable" "eth0=enable;eth1=enable" The following are the equivalent sysvar examples: sysvar set ipdhcpc6.if.eth0.dns enable sysvar set ipdhcpc6.if.eth0.dns enable sysvar set ipdhcpc6.if.eth1.dns enable	"" char *

Table 8-1 DHCPv6 Client Build Parameters (cont'd)

Workbench Description, Parameter Name, and sysvar	Default Value & Data Type
Interface SNTP Status List DHCPC6_IF_SNTP_LIST ipdhcpc6.if.interfaceName.sntp="status"	"" char *
<p>For interfaces that are able to receive stateless information, whether or not an individual interface can request information about available SNTP (Simple Network Time Protocol) servers. For each interface, enter either "enable" or "disable".</p> <p>The following are configuration parameter examples:</p> <pre>"eth0=enable" "eth0=enable;eth1=enable"</pre> <p>The following are the equivalent sysvar examples:</p> <pre>sysvar set ipdhcpc6.if.eth0.sntp enable sysvar set ipdhcpc6.if.eth0.sntp enable sysvar set ipdhcpc6.if.eth1.sntp enable</pre>	
Interface Information Refresh Status List DHCPC6_IF_INFO_REFRESH_LIST ipdhcpc6.if.interfaceName.information_refresh="status"	"" char *
<p>For interfaces that are able to receive stateless information, whether or not an individual interface requests a refresh time-interval from the DHCP server. The client uses the time-interval to determine how often it needs to refresh its stateless information. For each interface, enter either "enable" or "disable".</p> <p>The following are configuration parameter examples:</p> <pre>"eth0=enable" "eth0=enable;eth1=enable"</pre> <p>The following are the equivalent sysvar examples:</p> <pre>sysvar set ipdhcpc6.if.eth0.information_refresh enable sysvar set ipdhcpc6.if.eth0.information_refresh enable sysvar set ipdhcpc6.if.eth1.information_refresh enable</pre>	

Table 8-1 **DHCPv6 Client Build Parameters** (cont'd)

Workbench Description, Parameter Name, and sysvar	Default Value & Data Type
Interface Information Refresh Status List	""
DHCPv6_IF_INFO_REFRESH_MIN_LIST	char *
ipdhcpc6.if.interfaceName.information_refresh.minimum="time"	
For interfaces that are able to receive stateless information, the minimum time interval, in seconds, for a client to use in refreshing stateless information.	
The following are configuration parameter examples:	
eth0=900 eth0=900;vlan21=1200	
The following are the equivalent sysvar examples:	
sysvar set ipdhcpc6.if.eth0.information_refresh.minimum 900 sysvar set ipdhcpc6.if.eth0.information_refresh.minimum 900 sysvar set ipdhcpc6.if.vlan21.information_refresh.minimum 1200	
Interface Information Refresh Status List	""
DHCPv6_IF_INFO_REFRESH_DEFAULT_LIST	char *
ipdhcpc6.if.interfaceName.information_refresh.default="time"	
For interfaces that are able to receive stateless information, the default time interval, in seconds, for a client to use in refreshing stateless information.	
The following are configuration parameter examples:	
eth0=43200 eth0=43200;eth1=129600	
The following are the equivalent sysvar examples:	
sysvar set ipdhcpc6.if.eth0.information_refresh.default 43200 sysvar set ipdhcpc6.if.eth0.information_refresh.default 43200 sysvar set ipdhcpc6.if.eth1.information_refresh.default 129600	

Table 8-1 DHCPv6 Client Build Parameters (cont'd)

Workbench Description, Parameter Name, and sysvar	Default Value & Data Type
Interface Information Refresh Status List DHCPv6_IF_INFO_REFRESH_MAX_LIST ipdhcpc6.if.interfaceName.information_refresh.maximum="time"	"" char *
<p>For interfaces that are able to receive stateless information, the maximum time interval, in seconds, for a client to wait before refreshing stateless information. If no value is entered for an interface, there is no time limit on the interface.</p> <p>The following are configuration parameter examples:</p> <pre>"eth0=1728400" "eth0=1728400;eth1=1728400"</pre> <p>The following are the equivalent sysvar examples:</p> <pre>sysvar set ipdhcpc6.if.eth0.information_refresh.maximum 1728400 sysvar set ipdhcpc6.if.eth0.information_refresh.maximum 1728400 sysvar set ipdhcpc6.if.eth1.information_refresh.maximum 1728400</pre>	
Interface IA_NA Default List DHCPv6_IF_IA_NA_DEFAULT_LIST ipdhcpc6.if.interfaceName.ia_na.enum.default="status"	"" char *
<p>For interfaces that are able to receive stateful information, whether or not an individual interface is part of the default IANA.</p> <p>The Wind River Network Stack allows only one default Identity Association for Non-temporary Addresses (IANA) when you configure it statically. You can dynamically configure other IANAs.</p> <p>For each interface, enter either "enable" (meaning that the interface uses the default IANA) or "disable" (meaning that the default IANA is disabled, though you may still retrieve stateless information). The following are configuration parameter examples:</p> <pre>"eth0=enable" "eth0=enable;eth1=enable"</pre> <p>The following are the equivalent sysvar examples:</p> <pre>sysvar set ipdhcpc6.if.eth0.ia_na.enum.default enable sysvar set ipdhcpc6.if.eth0.ia_na.enum.default enable sysvar set ipdhcpc6.if.eth1.ia_na.enum.default enable</pre>	

Table 8-1 **DHCPv6 Client Build Parameters** (cont'd)

Workbench Description, Parameter Name, and sysvar	Default Value & Data Type
Interface IAID List	""
DHCPC6_IF_IA_NA_DEFAULT_IAID_LIST	
ipdhcpc6.if.interfaceName.ia_na.default.iaid="iaid"	char *
<p>This parameter assigns an Identity Association ID (IAID) to individual interfaces that are part of the default IANA. The IAID for each interface must be unique among all IAIDs for IANAs.</p> <p>For each interface, enter the interface name and IAID. The following are configuration parameter examples:</p> <pre>"eth0=1" "eth0=1;eth1=2;vlan21=3"</pre> <p>The following are the equivalent sysvar examples:</p> <pre>sysvar set ipdhcpc6.if.eth0.ia_na.default.iaid 1 sysvar set ipdhcpc6.if.eth0.ia_na.default.iaid 1 sysvar set ipdhcpc6.if.eth1.ia_na.default.iaid 2 sysvar set ipdhcpc6.if.vlan21.ia_na.default.iaid 3</pre>	
Interface Default Hints Status List	""
DHCPC6_IF_HINTS_DEFAULT_ENUM_LIST	
ipdhcpc6.if.interfaceName.ia_na.default.hints.enum.default="status"	char *
<p>For interfaces belonging to the default IANA, which interfaces can send hints to the DHCP server for preferred prefixes and timeout intervals.</p> <p>For each interface, enter either "enable" or "disable". The following are configuration parameter examples:</p> <pre>"eth0=enable" "eth0=enable;vlan2=disable"</pre> <p>The following are the equivalent sysvar examples:</p> <pre>sysvar set ipdhcpc6.if.eth0.ia_na.default.hints.enum.default enable sysvar set ipdhcpc6.if.eth0.ia_na.default.hints.enum.default enable sysvar set ipdhcpc6.if.vlan2.ia_na.default.hints.enum.default disable</pre>	

Table 8-1 DHCPv6 Client Build Parameters (cont'd)

Workbench Description, Parameter Name, and sysvar	Default Value & Data Type
Interface Prefix Hints List	""
DHCPC6_IF_HINTS_DEFAULT_PREFIX_LIST	char *
ipdhcpc6.if.interfaceName.ia_na.default.hints.default.prefix="prefix"	
For interfaces that can send hints (see the table entry for Interface Default Hints Status List), the preferred prefix for individual interfaces.	
For each interface, enter the interface name and the preferred prefix. The following are configuration parameter examples:	
<pre>"eth0=2001:DB8::"</pre> <pre>"eth0=2001:DB8::;eth1=2001:DB8:02::"</pre>	
The following are the equivalent sysvar examples:	
<pre>sysvar set ipdhcpc6.if.eth0.ia_na.default.hints.default.prefix 2001:DB8::</pre> <pre>sysvar set ipdhcpc6.if.eth0.ia_na.default.hints.default.prefix 2001:DB8::</pre> <pre>sysvar set ipdhcpc6.if.eth1.ia_na.default.hints.default.prefix 2001:DB8:02::</pre>	
Interface Hints Valid List	""
DHCPC6_IF_HINTS_DEFAULT_VALID_LIST	char *
ipdhcpc6.if.interfaceName.ia_na.default.hints.default.valid="time"	
For interfaces that can send hints (see the table entry for Interface Default Hints Status List), the <i>valid lifetime</i> (see RFC 2462) that an individual interface wants the server to assign to IPv6 addresses.	
For each interface, enter the interface name and a lifetime. The lifetime specified for an interface must be greater than or equal to the interface's <i>preferred lifetime</i> (see the table entry for Interface Preferred List). The following are configuration parameter examples:	
<pre>"eth0=6000"</pre> <pre>"eth0=6000;eth1=5000;vlan21=7000"</pre>	
The following are the equivalent sysvar examples:	
<pre>sysvar set ipdhcpc6.if.eth0.ia_na.default.hints.default.valid 6000</pre> <pre>sysvar set ipdhcpc6.if.eth0.ia_na.default.hints.default.valid 6000</pre> <pre>sysvar set ipdhcpc6.if.eth1.ia_na.default.hints.default.valid 5000</pre> <pre>sysvar set ipdhcpc6.if.vlan21.ia_na.default.hints.default.valid 7000</pre>	

Table 8-1 **DHCPv6 Client Build Parameters** (cont'd)

Workbench Description, Parameter Name, and sysvar	Default Value & Data Type
Interface Preferred List	""
DHCPC6_IF_HINTS_DEFAULT_PREFERRED_LIST	
ipdhcpc6.if.interfaceName.ia_na.default.hints.default.preferred ="time"	char *
<p>For interfaces that can send hints (see the table entry for Interface Default Hints Status List), the <i>preferred lifetime</i> (see RFC 2462) that an individual interface wants the server to assign to IPv6 addresses.</p> <p>For each interface, enter the interface name and a preferred lifetime, in seconds. The preferred lifetime specified for an interface must be less than or equal to the interface's <i>valid lifetime</i> (see the table entry for Interface Hints Valid List). The following are configuration parameter examples:</p> <pre>"eth0=7000" "eth0=7000;eth1=6000;vlan21=8000"</pre> <p>The following are the equivalent sysvar examples:</p> <pre>sysvar set ipdhcpc6.if.eth0.ia_na.default.hints.default.preferred 7000 sysvar set ipdhcpc6.if.eth0.ia_na.default.hints.default.preferred 7000 sysvar set ipdhcpc6.if.eth1.ia_na.default.hints.default.preferred 5000 sysvar set ipdhcpc6.if.vlan21.ia_na.default.hints.default.preferred 8000</pre>	

Table 8-1 DHCPv6 Client Build Parameters (cont'd)

Workbench Description, Parameter Name, and sysvar	Default Value & Data Type
Interface Renew List	""
DHCPv6_IF_HINTS_DEFAULT_RENEW_LIST	char *
ipdhcpc6.if.interfaceName.ia_na.default.hints.renew="time"	
<p>For interfaces that can send hints (see the table entry for Interface Default Hints Status List), the amount of time individual interfaces want the server to add to their valid and preferred lifetimes before the server requires address renewal.</p> <p>For each interface, enter the interface name and a renewal extension time, in seconds. The following are configuration parameter examples:</p> <pre>"eth0=7000" "eth0=7000;eth1=6000;vlan21=8000"</pre> <p>The following are the equivalent sysvar examples:</p> <pre>sysvar set ipdhcpc6.if.eth0.ia_na.default.hints.renew 7000 sysvar set ipdhcpc6.if.eth0.ia_na.default.hints.renew 7000 sysvar set ipdhcpc6.if.eth1.ia_na.default.hints.renew 6000 sysvar set ipdhcpc6.if.vlan21.ia_na.default.hints.renew 8000</pre>	
DHCPv6 Client Interface Rebind List	""
DHCPv6_IF_HINTS_DEFAULT_REBIND_LIST	char *
ipdhcpc6.if.interfaceName.ia_na.default.hints.rebind="time"	
<p>For interfaces that can send hints (see the table entry for Interface Default Hints Status List), a desired rebind time for individual interfaces. The rebind time is similar to the renewal extension time (see the table entry for Interface Renew List), but applies only when the client has sent a Renew message without receiving a response (see RFC 3315).</p> <p>For each interface, enter the interface name and a rebind time, in seconds. The following are configuration parameter examples:</p> <pre>"eth0=7000" "eth0=7000;eth1=6000;vlan21=8000"</pre> <p>The following are the equivalent sysvar examples:</p> <pre>sysvar set ipdhcpc6.if.eth0.ia_na.default.hints.rebind 7000 sysvar set ipdhcpc6.if.eth0.ia_na.default.hints.rebind 7000 sysvar set ipdhcpc6.if.eth1.ia_na.default.hints.rebind 6000 sysvar set ipdhcpc6.if.vlan21.ia_na.default.hints.rebind 8000</pre>	

8.3 Using Shell Commands

You can use shell commands to do the following:

- View the current configuration of client interfaces.
- Refresh the configuration of client interfaces.
- Release the current configuration of client interfaces.

All DHCPv6 client shell commands start with the key word **dhcpc6** and have the form:

```
dhcpc6 [-v] command [interfaceName [iana [ianaName]]]
```

The parameters are as follows:

- **-v** turns on verbose reporting.
- *command* is one of the shell commands described below.
If the command is not followed by an interface specification, the command applies to all interfaces.
- *interfaceName*, if specified, is the name of an interface to which the command applies.
- *iana*, if specified, means that the command applies only to the non-temporary identity associations (IANAs) bound to the specified interface.
- *ianaName*, if specified, is the name of the IANA to which the command applies.

The individual commands are as follows:

dhcpc6 show

Show the current configuration of all DHCPv6 interfaces or of a specified interface. For example, to show the configuration of interface eth0, with verbose reporting turned on, enter:

```
-> dhcpc6 -v show eth0
```

The following is sample output from a **dhcpc6 show** command:

```
-> dhcpc6 show vlan100 ia_na default
Interpeak DHCPv6 Client:
DUID      : 06:30:00:10:00:00:00:00:02
Interface: vlan100
Mode      : enabled
Link State : up
Non-Temporary Identity Association default, 1:
Status    : bound/idle
```

```
Binding:
  DHCP Server : 2001:DB8::200:FF:FE00:3
  Renew/Rebind: 10/16
  Lease       : 2001:DB8:2::D/64 preferred 20 valid 40
```

`dhcpc6 history show`

Show the history of all DHCPv6 interfaces or of a specified interface. For example, to show the history of all client interfaces, enter:

`-> dhcpc6 history show`

The following is sample output from this shell command:

```
-> dhcpc6 history show vlan100 ia_na default
Interpeak DHCPv6 Client:
DUID       : 06:30:00:10:00:00:00:00:02
Interface: vlan100
Non-Temporary Identity Association default, 1:
History(6):
  Sun Feb 18 21:32:11 2007: solicit/idle
  Sun Feb 18 21:32:11 2007: solicit/solicit
  Sun Feb 18 21:32:11 2007: solicit/soliciting
  Sun Feb 18 21:32:14 2007: solicit/request
  Sun Feb 18 21:32:14 2007: solicit/requesting
  Sun Feb 18 21:32:14 2007: bound/idle
```

`dhcpc6 history flush`

Delete the history of all DHCPv6 interfaces or of a specified interface. The following example deletes the history of the default IANA on interface `vlan100`:

`-> dhcpc6 history flush vlan100 ia_na default`

`dhcpc6 refresh`

Request confirmation of current configuration information. If confirmation is not received, the command tries to acquire information from the network. The following example requests confirmation of non-temporary information on interface `eth0`, restricted to the default IANA.

`-> dhcpc6 refresh eth0 ia_na default`

`dhcpc6 release`

Releases current configuration information for all interfaces or for a specified interface. For example, to release all configuration information, enter:

`-> dhcpc6 release`

9

Creating Network Applications as RTPs

- 9.1 Introduction 135
- 9.2 Running Network Applications in RTPs 136
- 9.3 Working with Application RTPs 137
- 9.4 Using Socket Connections with RTPs 142

9.1 Introduction

A Real Time Process (RTP) is one that runs in isolation from the VxWorks kernel, with its own symbol namespace and memory region.

VxWorks once only allowed kernel execution mode. Both user applications and central functionality, such as that provided by the network stack, ran in the same memory space. Now VxWorks has an RTP mode in addition to kernel mode. This RTP mode is based on RTP Executable and Linking Format (ELF) object files. These object files are fully-linked, relocatable executables with full name space isolation. Using an RTP ELF object, you can isolate an application from the kernel and from applications running in other RTPs. This isolation of an application to an RTP ELF object allows for memory protection.

You do not have to run an application as an RTP. You can still run it in the kernel. You are also free to run some applications in the kernel while others execute as RTPs.

The core network stack functionality, which consists of fundamental networking protocols such as IP, TCP, UDP, the network interface drivers, and the MUX functionality, runs in the kernel only. If your network application relies on direct access to that functionality, it cannot be written as an RTP.

The standard socket APIs and the routing socket APIs are available to RTPs. If your application limits (or can limit) its interaction with the network stack to standard or routing socket API calls, your application is a good candidate to run in an RTP.

This chapter shows how to modify an existing network application to run as an RTP. This chapter also shows how RTPs can use sockets.

For more information on RTPs, and on RTP projects in Workbench, see the *Wind River Workbench User's Guide*.

9.2 Running Network Applications in RTPs

Many network applications use a client/server model. The server side of the application is typically implemented as a daemon that runs in the context of its own independent task. The client side of the application is typically implemented as a library whose routines execute in the context of the caller.

The network stack includes a sample implementation of a ping client as an RTP, in the form of a **.vxe** file.

Running Ping in an RTP

To load the **ping.vxe** RTP, call **rtpSp()**. From a kernel shell, **cd** to the directory containing the **ping.vxe** and issue the following command:

```
-> rtpSp "ping.vxe remoteSystem numberOfPackets options"
```

ping.vxe initializes the **ping** library, handles the **ping** request for the requested number of packets, and exits. When **ping** exits, this shuts down the RTP. If you specify a continuous **ping** (a **ping** with a *numberOfPackets* value of 0), **ping** runs forever, or until you shut it down explicitly.

9.2.1 General Network/RTP Incompatibilities

Core network stack protocols such as IP, TCP, UDP, and the like run in kernel space only, although RTPs may access them through socket connections and `sysctl()` calls.

The MUX functionality and network interface drivers also run in kernel space only. If you want to launch and configure a network interface, do so within the kernel space. RTPs cannot create and configure network interfaces.

You can call most network show and configuration utilities from kernel space only. The exception is the `sysctl()` routine, which RTPs may call.

Because of the isolation of an RTP memory space, RTPs cannot supply callback routines to kernel-resident network protocols and services. A pointer to a routine in an RTP memory space is meaningless in the kernel memory space.

9.3 Working with Application RTPs

All applications, whether running in the kernel or as RTPs, can communicate with each other using `AF_LOCAL` sockets. In addition, RTP-based applications can access kernel-resident services using `sysctl()` or system calls. If you add a new kernel-resident service, you can extend the set of system calls to include that service.

The following sections give an overview of how to work with RTPs. Included are brief discussions of how to write, build, and launch an RTP. More information is available in the *VxWorks Programmer's Guide*.

9.3.1 Building an RTP ELF Object File for a Network Application

This section describes how to write and build a “Hello World” application to run as an RTP. Although “Hello World” is not particularly representative of a network application, porting “Hello World” to an RTP demonstrates the most basic mechanics of creating an RTP.

Example 9-1 **Writing the Code for a “Hello World” RTP**

Consider the “Hello World” C program:

```
int main
(
    int    argc,          /* number of arguments */
    char * argv[],        /* array of arguments */
)
{
    printf ("hello world\n");
    return 0;
}
```

To create an RTP that uses this **main()**, you need to change it slightly to accept all the inputs provided when an RTP executes its **main()**.

```
int main
(
    int    argc,          /* number of arguments */
    char * argv[],        /* array of arguments */
    char * envp[],        /* array of environment strings */
    void * auxp           /* implementation-specific auxiliary vector */
)
{
    printf ("hello world\n");
    return 0;
}
```

The **argc** and **argv** parameters serve their traditional function of passing in a count of caller arguments as well as an array containing the arguments themselves. You can use **envp** and **auxp** parameters to pass in whatever additional information your RTP needs at run-time to configure its environment or any auxiliary implementation-specific need.

Example 9-2 **Building the Code for a “Hello World” RTP**

To build the code given in [Example 9-1](#) as an RTP ELF object file, you need a makefile with the following basic format:

```
# Makefile - makefile for hello world example RTP
#
# DESCRIPTION
# This file contains the makefile rules for building an example RTP

EXE = helloworld.vxe
OBJS = mainHello.o
include $(WIND_USR)/make/rules.rtp
```

This makefile assumes that the “Hello World” **main()** is defined in a file called **helloworld.c**. To build the **helloworld.vxe**, you would run the following **make**:

```
$ make CPU=cpuType TOOL=toolChain
```

where:

cpuType is a valid CPU type, such as **PPC32**.

toolChain is a value such as **diab** or **gnu**

Setting up Pre-Entry-Point Initialization Routines

What if “Hello World” were not written to print the message but to use a customized service to transmit the message over the Internet? Such services typically require some initialization that must complete before an application can use the service. Some libraries can instruct the system to automatically call their initialization routine by converting their initialization routine into an RTP constructor. For more information, see [9.3.3 Identifying the RTP Constructor Routine in a Library](#), p.140.

Some utilities and services have initialization routines that must obtain information at run-time (for instance, the IP address of a server). Such initialization routines cannot be converted into RTP constructors. If your utility or service has this sort of initialization routine, write your RTP **main()** to call the initialization routine directly.

9.3.2 Launching an RTP

There are several ways to launch an RTP. From a kernel shell, you can use **rtpSp()**:

```
-> rtpSp "ELFobjectFile [arg1 [...arg20]]"
```

For example:

```
-> rtpSp "path/ping.vxe remoteSystem numberOfPackets options"
```



NOTE: The **rtpSp()** utility works in the kernel shell only, not the host shell.

If you use the command interpreter, you can omit the explicit call to **rtpSp()**. For example:

```
# path/ping.vxe remoteSystem numberOfPackets options
```

Alternatively, you can use **rtpSpawn()**, which is called internally by **rtpSp()**. **rtpSpawn()** gives you fine-grain control of RTP creation: It lets you specify the **main()** task priority, its stack size, and supplemental data that the RTP can use to configure its environment and the application it runs. Within an **rtpSp()** call, the task priority is set to 220, and the stack size set to 40K bytes. If these default values are not appropriate for your RTP, call **rtpSpawn()** instead. **rtpSpawn()** is a system call and can be called from outside the kernel.

Calling `rtpSpawn()`

Because `rtpSpawn()` is implemented in the kernel as a system call, you can call it from code running either in the kernel or in a previously launched RTP.

`rtpSpawn()` validates and then loads the RTP object file. The RTP then executes all the initialization routines for libraries that the RTP references. Within each library, the `_WRS_CONSTRUCTOR` attribute indicates the RTP initialization routine for the library.

After validating, loading, and initializing the RTP, `rtpSpawn()` launches it as an autonomous task. The task routine for this task is the `main()` provided by the RTP ELF object file, named in the `rtpFileName` parameter to `rtpSpawn()`. Barring an external shutdown, the life span of an RTP continues until any task within the RTP calls `exit()` or until execution reaches the end brace of the RTP's `main()`.

Within the RTP `main()`, use the standard `taskSpawn()` and `taskCreate()` routines to launch any additional tasks your application requires. These tasks run in the context of the RTP and cannot survive the expiration of the RTP. Write these tasks so that they shut themselves down with `taskExit()`, since this routine ends only the task that calls it. A task that calls `exit()` from within an RTP shuts down the entire RTP and *all* of its tasks.

9.3.3 Identifying the RTP Constructor Routine in a Library

If an RTP references any libraries in its `main()` routine, either directly or indirectly, when the RTP first launches (after `rtpSpawn()` validates it) it runs the constructor routines for those libraries. When the RTP is built, the system automatically generates this list of constructor routines by processing the libraries that the RTP references and making note of those library routines that are marked with the `_WRS_CONSTRUCTOR` attribute. When you launch the RTP, the system calls each of these constructor routines in an order that is determined by the priority in their `_WRS_CONSTRUCTOR` attribute.

After these constructor routines return successfully, the RTP executes the RTP's `main()`.



NOTE: Initialization routines that require run-time input cannot be RTP constructors. You must call such initialization routines from the RTP `main()` or from a child task launched from the RTP's `main()`.

If you have written a library whose initialization routine you want the system to call automatically as an RTP constructor, the syntax is as follows:

```
_WRS_CONSTRUCTOR ( functionName, priority )
```

functionName

The name of the initialization routine.

priority

Determines the execution order of the initialization routines for an RTP. The lower the priority value, the higher the priority given to the routine. For example, an initialization routine with a priority of 10 executes before an initialization routine with a priority of 20. If two initialization routines have the same priority, the execution order of those same-priority routines is arbitrary. If one routine must execute before another, adjust the priority values appropriately.

For an example of how to use `_WRS_CONSTRUCTOR`, consider the hypothetical initialization routine, `myLibInit()`:

```
STATUS myLibInit (void)
{
    ...
}
```

To make this initialization routine behave as an RTP constructor routine, rewrite its declaration using the `_WRS_CONSTRUCTOR` format (and assign the constructor a priority: in this example, 5):

```
_WRS_CONSTRUCTOR (myLibInit, 5)
{
    ...
}
```

The Wind River Compiler and the GNU compiler both have this mechanism.

9.3.4 Shutting down an RTP Application

An RTP shuts down entirely when any task in the RTP calls `exit()`. To avoid accidentally shutting down an RTP, you need to be careful about when you call `exit()`. You also need to avoid implicit calls to `exit()`. These occur if the execution reaches the end brace of the `main()` routine.

One way to avoid an implicit `exit()` call is to call `taskExit()` before execution reaches the routine end brace. As an alternative to `taskExit()`, you could call:

```
taskSuspend(0);
```

In most situations, a `taskExit()` call is the more elegant solution.

To abort an RTP from the outside, call `rtpDelete()`. As input, this routine requires an `RTP_ID` that identifies the RTP that you want to delete. You receive this `RTP_ID`

as the returned value of the **rtpSpawn()** call that launches the RTP. You can also get this ID by calling **rtpShow()**, which includes RTP IDs in its report.

As an alternative to **rtpDelete()**, you can use **rtpKill()** to send a **SIGKILL** (9) or a **SIGTERM** (15) signal to the RTP. If the application installed a signal handler for **SIGTERM**, a **SIGTERM** lets the application shut itself down in a controlled manner. If the application has not installed a signal handler for **SIGTERM**, either **SIGKILL** or **SIGTERM** abort the application.

9.4 Using Socket Connections with RTPs

For many network applications, BSD sockets in the Internet domain provide the fundamental communication mechanism. The protocols and services that enable Internet sockets reside in the kernel. If your application runs in the kernel, it can access these socket services directly.

To make these services available to applications running as RTPs, the kernel exports a set of routines as system calls. This set of socket-related calls includes all the well-known routines in the BSD socket API plus the VxWorks-specific **connectWithTimeout()** routine. See [Table 9-1](#).

Table 9-1 **Socket Calls**

Routine Name	Use
socket()	Create an end point for communication.
bind()	Associate a local address with a socket.
listen()	Mark a socket as accepting connections.
accept()	Accept a connection on a socket.
connect()	Initiate a connection on a socket.
sendto()	Send a message from a socket.
sendmsg()	Send a message from a socket using a structure.
send()	Send a message from a connected socket.

Table 9-1 Socket Calls (cont'd)

Routine Name	Use
recvfrom()	Receive a message from a socket and capture the address of the sender.
recv()	Receive a message from a socket.
recvmsg()	Receive a message from a socket and store it in a structure.
setsockopt()	Set socket options.
getsockopt()	Get socket options.
getsockname()	Get the socket name.
getpeername()	Get the name of the peer that is connected to the socket.
shutdown()	Shut down the full-duplex connection on the socket.

Using Sockets for Inter-Task Communication

In addition to Internet domain sockets, VxWorks allows local domain (**AF_LOCAL**) sockets for communication among tasks, even tasks that execute in different memory spaces (the kernel memory space and the various RTP memory spaces). Under VxWorks, **AF_LOCAL** sockets allow only one set of communications characteristics: bidirectional, reliable, sequenced, non-duplicated, and *packet-based*. This socket communications style type is called **SOCK_SEQPACKET**.

For more information on **AF_LOCAL** sockets, see [10. Internet and Local Domain Sockets](#).

10

Internet and Local Domain Sockets

10.1 Introduction	145
10.2 Configuring VxWorks for Sockets	147
10.3 Using Sockets in VxWorks	150
10.4 Working with Local Domain Sockets	154
10.5 Working with Internet Domain Sockets	156

10.1 Introduction

This chapter describes how to use the standard socket interface for Internet domain sockets on a target running the Wind River Network Stack. Some mention is made of routing sockets and local domain sockets.

For detailed information on the following related topics, refer to the appropriate chapters:

- using sockets in RTPs

- See *9.4 Using Socket Connections with RTPs*, p.142.

- using routing sockets

- See *Wind River Network Stack for VxWorks 6 Programmer's Guide, Volume 1: Transport and Network Protocols*.

adding socket-support code to a new network service or protocol

See *Wind River Network Stack for VxWorks 6 Programmer's Guide, Volume 3: Integrating a New Network Service*.

Essential Background Reading

This chapter introduces the particulars of the Wind River Network Stack sockets implementation. A discussion of how to work with sockets in general is beyond the scope of this chapter.

The basic standard sockets API is documented as part of POSIX in IEEE Std. 1003.1, with more advanced features documented in various RFCs (see [Conformance to Standards](#), p.146). For general socket programming information, see *UNIX Network Programming, Volume 2, Second Edition* by W. Richard Stevens.

VxWorks Environment-Specific Socket Issues

Although the Wind River Network Stack is for the most part compatible with the BSD socket interface, the particulars of the VxWorks environment do affect how you use sockets.



WARNING: Because there are globally accessible file descriptors in the task-independent address space of the VxWorks kernel or an RTP, you must take precautions when closing a file descriptor (a socket descriptor is a variety of file descriptor). In the VxWorks kernel, one task must not close a file descriptor that another task is using, as this is not a safe operation. As a result of the close done by a different task, an operation in progress on the descriptor might return an error but also might experience an exception or other ill effects. Also, if an operation is performed on the descriptor after it is closed, the operation would normally return an error, but it could also affect another socket or file if the system reuses the descriptor after the descriptor is closed.

To wake up a task that is blocked on a socket, either use **shutdown()**, set a timeout on the socket function on which the task is blocked, or deliver an event (for example, a message) over the socket on which the task is blocked. You can also set up a semaphore-based locking mechanism that prevents the close while an operation is pending on the descriptor.

Conformance to Standards

For IPv6 sockets in particular, read RFC 3493: *Basic Socket Interface Extensions for IPv6* and RFC 3542: *Advanced Sockets API for IPv6*. The Wind River Network Stack Socket implementation implements these RFCs. It does not now implement any proposals published only in the Internet drafts.



NOTE: The update to RFC 3542 has changed some of the definitions in a way that is not backward compatible. The IPv6 APIs in the network stack have changed to match the updated RFC. For more information, see the release notes and RFC 3542.

Include Files Referenced in this Chapter

This chapter makes reference to the include files, **socket.h**, **in.h**, **in6.h**, and **tcp.h**. For components running in the kernel, the full pathnames of these files are as follows:

- *installDir/vxworks-6.5/target/h/wrn/coreip/sys/socket.h*
- *installDir/vxworks-6.5/target/h/wrn/coreip/netinet/in.h*
- *installDir/vxworks-6.5/target/h/wrn/coreip/netinet6/in6.h*
- *installDir/vxworks-6.5/target/h/wrn/coreip/netinet/tcp.h*

For components running in RTPs, the full pathnames of the files are as follows :

- *installDir/vxworks-6.5/target/usr/h/wrn/coreip/sys/socket.h*
- *installDir/vxworks-6.5/target/usr/h/wrn/coreip/netinet/in.h*
- *installDir/vxworks-6.5/target/usr/h/wrn/coreip/netinet6/in6.h*
- *installDir/vxworks-6.5/target/usr/h/wrn/coreip/netinet/tcp.h*

10

10.2 Configuring VxWorks for Sockets

The Wind River Network Stack uses the following configuration components for sockets:

- **Network Sockets (Folder)**
 - **Netlink socket** (INCLUDE_IPNET_USE_NETLINKSOCK)
 - **Socket backend** (INCLUDE_IPNET_USE_SOCKET_COMPAT)
 - **Socket support** (INCLUDE_IPNET_SOCKET)
 - **routing socket support** (INCLUDE_IPNET_USE_ROUTE SOCK)
- **Socket API** (INCLUDE_SOCKETLIB)
- **Socket API System Call support** (INCLUDE_SC_SOCKETLIB)

The Socket API (extensible interface) is the core component within this system. It enables you to implement your own socket domain handler and mediates access to registered socket back ends.



NOTE: This section focuses on components associated with network-related sockets (that is, Internet sockets and routing sockets). Not described here is the **INCLUDE_UN_COMP** component, which pulls in an implementation of the COMP protocol. This protocol is required for **AF_LOCAL** domain **SOCK_SEQPACKET** type sockets.

Netlink socket (INCLUDE_IPNET_USE_NETLINKSOCK)

Netlink sockets are not supported on VxWorks, but only in Linux Kernel Mode.

Socket backend (INCLUDE_IPNET_USE_SOCKET_COMPAT)

Include this component in any build that uses the socket API. Along with the **Socket support** component, it provides the standard socket back end to the socket API.

Socket support (INCLUDE_IPNET_SOCKET)

If you want to use sockets, you need to include this component in your build. This is the standard socket implementation for the network stack. These sockets have the same semantics and use the same structures as the BSD 4.4 socket API.

routing socket support (INCLUDE_IPNET_USE_ROUTE SOCK)

Routing sockets are fully described in the *Routing Sockets* chapter of *Wind River Network Stack for VxWorks 6 Programmer's Guide, Volume 1: Transport and Network Protocols*.

Socket API (INCLUDE_SOCKETLIB)

If you write a socket application you must add this component to your build. This component pulls in **sockLib**, a VxWorks library that implements the standard socket interface layer. This layer mediates access to registered socket back ends. If you include any of the other sockets components, this automatically registers their socket back ends with the standard socket interface layer.

For information on how to register additional socket back ends, see *Wind River Network Stack for VxWorks 6 Programmer's Guide, Volume 3: Integrating a New Network Service*, and the **sockLib** reference entry.

Socket API System Call support (INCLUDE_SC_SOCKETLIB)

You must include this component in your build if your build contains a real-time process (RTP) that uses the socket API. This component pulls in

sockScLib, a collection of socket API stubs that allow RTPs to use socket calls even if the RTPs are in a different address space from the network stack.

This component is included automatically when RTP support, **INCLUDE_RTP**, is included, and this component also requires the **INCLUDE_RTP** component.

The parameters used to configure sockets are described in [Table 10-1](#).

Table 10-1 **Socket Configuration Parameters**

Workbench Description and Parameter Name	Default Value and Type
Address Notify IPNET_SOCK_ADDR_NOTIFY If you set this to "1", when an address is removed from a node on which socket applications are running, the stack sends an ENETDOWN failure message to all socket owners that bound their sockets to that address and that are blocked on an accept() call. Set this to "0" to turn off this notification, in which case the socket owners will remain blocked on accept() in such a circumstance.	"1" char *
AnonPortMax IPNET_SOCK_ANON_PORT_MAX The highest port number that the system may use as an ephemeral port number. The system assigns a port in the ephemeral port number range from AnonPortMin to AnonPortMax to a socket if that socket does not select its own port with a bind() call.	"65535" char *
AnonPortMin IPNET_SOCK_ANON_PORT_MIN The lowest port number that the system may use as an ephemeral port number. The system assigns a port in the ephemeral port number range from AnonPortMin to AnonPortMax to a socket if that socket does not select its own port with a bind() call.	"49152" char *

Table 10-1 **Socket Configuration Parameters** (cont'd)

Workbench Description and Parameter Name	Default Value and Type
Default socket receive buffer size IPNET_SOCK_DEFAULT_RECV_BUFSIZE The default size of the socket receive buffer.	"10000" char *
Default socket send buffer size IPNET_SOCK_DEFAULT_SEND_BUFSIZE The default size of the socket send buffer.	"10000" char *
Maximum number of sockets IPNET_SOCK_MAX The maximum number of sockets in use. Note that because a socket descriptor is a variety of file descriptor, the number of sockets is also limited by the maximum number of file descriptors, NUM_FILES, which defaults to 50.	1024 uint

10.3 Using Sockets in VxWorks

The Wind River Network Stack includes a standard socket interface to TCP and UDP. Using sockets, you can do the following:

- Send and receive data over an IP network.
- Communicate with other processes.
- Access IP multicasting functionality.
- Review and modify the routing tables.

A socket is a communications endpoint to which you can bind an address. You use this address to access the socket. Implicit in the idea of an endpoint is the notion of a space that contains the endpoint. By convention, the space containing a socket is called a communications domain. Valid addresses for a socket are defined in terms of the socket's communication domain.

This definition of a socket indicates that the socket metaphor is not necessarily limited to a programming environment. For example, you can think of a telephone as a socket. The telephone is a communications endpoint to which the phone

company has bound an address: a telephone number. This telephone number has been chosen from the communications domain of valid telephone numbers.

Within a programming environment, a socket is often implemented as a descriptor to which you bind an address. Associated with the descriptor are routines that can read or write information to or from the descriptor. The type of address that you can assign to the descriptor depends on the communications domain for the socket. For more information on socket types, see [Socket Types](#), p.153.

Communications Domains

BSD UNIX has a long list of domains for its sockets. Of these, VxWorks has only the following:

- the IPv4 Internet domain (**AF_INET**)
- the IPv6 Internet domain (**AF_INET6**)
- the routing domain (**AF_ROUTE**)
- local domain sockets for inter-process communication (**AF_LOCAL**)
- the Transparent Inter-Process Communication domain (**AF_TIPC**)
- the Mobile IPv6 domain (**AF_MOBILITY**)



NOTE: Neither the **AF_TIPC** domain sockets nor the **AF_MOBILITY** domain sockets are described in this manual. For information on **AF_TIPC** sockets, see the *Wind River TIPC for VxWorks 6 Programmer's Guide*. For information on **AF_MOBILITY** sockets, see the *Wind River Network Stack for VxWorks 6 Programmer's Guide, Volume 1: Transport and Network Protocols*.

Internet Domain Sockets

Several operating systems use Internet domain socket connections to communicate. Applications typically use sockets in the Internet domains to exchange information with peers on remote host systems. You can also use sockets in these domains to add or remove routing entries in the stack's multicast forward information base (routing table).

In the Internet domains, the VxWorks socket support functions let you define socket endpoints using either IPv4 or IPv6 addresses.

- Sockets in the IPv4 domain, **AF_INET**, bind to names defined in terms of an IPv4 address and a port number.
- Sockets in the IPv6 domain, **AF_INET6**, bind to names defined in terms of an IPv6 address and a port number, and in some cases, a scope identifier.

Applications typically use sockets in the Internet domains to exchange information with peers on remote host systems, but you can also use an Internet domain socket to communicate with a peer on the local host. However in VxWorks there are more efficient mechanisms for communication between local tasks, such as **AF_LOCAL** sockets, TIPC, and message queues.

As much as possible, the socket API is IPv6/IPv4 agnostic. Thus, the **sockLib** calls have been modified internally to respond appropriately to an IPv6 address, but they do this only if the **socket()** call that created the socket specified the IPv6 address family. In addition, to pass in an IPv6 address, you need to use a **sockaddr_in6** structure that you have cast to a **sockaddr** structure.

Despite this adaptation to IPv6 addresses, there are some socket-mediated services that will not work with IPv6 without modification. The IPv4 multicast application in particular must be rewritten in order to support IPv6 multicasting, since that part of the socket API differs substantially between the two protocols.

For more information on Internet domain sockets, see [10.5 Working with Internet Domain Sockets](#), p. 156.

Routing Sockets

Sockets in the routing domain, **AF_ROUTE**, communicate with the local route table. Use a routing socket to make or to monitor changes to the route table.

For a description of how routes are ranked in the route table, and for more information on routing sockets, see *Wind River Network Stack for VxWorks 6 Programmer's Guide, Volume 1: Transport and Network Protocols*.

Local Domain Sockets

Sockets in the local domain, **AF_LOCAL**, bind to names modeled after the names of files in a file system (no actual file system is necessary to use **AF_LOCAL** sockets). In VxWorks, local domain socket names are implemented over Wind River's Connection-Oriented Message Passing (COMP) protocol. This protocol provides an optimized transport (better performance than **AF_INET** or **AF_INET6**) for applications that use the socket API to communicate between tasks that are running on the same node. This method is able to communicate between RTPs and the kernel, or among tasks running in different RTPs.



NOTE: The domain handler for the **AF_LOCAL** domain, COMP, is independent of the network stack. If the only sockets you need to use are **AF_LOCAL** sockets, you do not need to include a network stack in your VxWorks image.

For more information on local domain sockets, see [10.4 Working with Local Domain Sockets](#), p.154.

Raw Sockets

Raw sockets bypass the transport layer, which allows applications that use raw sockets to implement their own transport-layer processing from outside of the stack.

Socket Types

A socket's type describes the communication style that the socket uses. VxWorks includes the following socket communication styles or types:

SOCK_DGRAM

bidirectional, unreliable, not sequenced, possibly duplicated, message oriented

SOCK_STREAM

bidirectional, reliable, sequenced, non-duplicated, stream oriented

SOCK_SEQPACKET

bidirectional, reliable, sequenced, non-duplicated, message oriented

SOCK_RAW

usually bidirectional, interface- and protocol-dependent; provides access to internal network protocols and interfaces

SOCK_RDM

bidirectional, reliable, not sequenced, possibly duplicated, message oriented. The Wind River Network Stack allows this type for **AF_TIPC** sockets only. For more information, see *Wind River TIPC for VxWorks 6 Programmer's Guide*.

The **SOCK_DGRAM** type is a connectionless communication style based on an exchange of datagrams. This style of communication is analogous to an exchange of letters (datagrams) using a postal service. The same datagram socket may communicate with several peers. Any acknowledgement that a datagram is received is the responsibility of the application, rather than the underlying protocol: Datagram sockets do not automatically retransmit unacknowledged datagrams. They also do not guarantee that they will deliver datagrams in order, or that they will deliver a particular datagram only once. Internet domain datagram sockets operate over UDP, a transport layer protocol of the IP stack.

The **SOCK_STREAM** type is a connection-oriented communication style characterized by the exchange of a continuous stream of data. This style of

communication is analogous to a phone conversation. A **SOCK_STREAM** socket establishes a connection with a single peer. Data are sent as a stream of bytes that are delivered in order to the peer. If any segment of the stream fails to be acknowledged by its destination, the protocol will retransmit that segment. If, in spite of retransmission attempts, the recipient fails to acknowledge the data, the connection terminates and notifies the sender. Internet domain stream sockets operates over TCP, a transport layer protocol of the IP stack.

The **SOCK_SEQPACKET** type describes a communication style that is both connection oriented and message oriented. Being message oriented means that recipients of data over **SOCK_SEQPACKET** sockets receive data as discrete messages instead of as a stream of data. The Wind River Network Stack allows this socket type for sockets in the **AF_LOCAL** domain only.

The **SOCK_RAW** socket is a socket with an undefined communication style. Its communication characteristics (reliable or not, sequenced or not, and so on) are determined by the services provided by the domain and protocol that you specify when you create the socket. For example, an application might create a raw **AF_INET** or **AF_INET6** socket and implement SCTP over it.

In the Wind River Network Stack, the primary use of **SOCK_RAW** sockets is within the **AF_ROUTE** domain, where tasks can use sockets to monitor and make changes in the contents of the local route table.

10.4 Working with Local Domain Sockets

Under VxWorks, local domain sockets (**AF_LOCAL** sockets) are based on a protocol called COMP. This protocol enables the **SOCK_SEQPACKET** communication style, which is bidirectional, reliable, and message-based.

The Wind River Network Stack implements the standard socket APIs such as **socket()**, **bind()**, and **listen()**. It also implements the standard file system APIs: **read()**, **write()**, **close()**, **ioctl()**, and **select()**.

Because the **AF_LOCAL/COMP** sockets use a connection-oriented communication style, you must complete a bind-listen-connect-accept sequence before you can transmit data on the socket. To transmit and receive data over the socket, use the **send()** and **recv()** routines. The Wind River Network Stack implements the **setsockopt()**, **getsockopt()**, and **ioctl()** routines, but with a minimal set of options.

Including Local Domain Sockets

To create a VxWorks image that includes local domain sockets, you must include the **INCLUDE_UN_COMP** component in your build. This component pulls in the COMP implementation. Note that this component is not a network stack component and that COMP is independent of the network stack. You can include COMP in a VxWorks image and use **AF_LOCAL** sockets even if you do not include a network stack.

Setting up a Local Domain Socket

Under VxWorks, COMP manages the name space for an **AF_LOCAL** domain socket. Thus, when creating names for **AF_LOCAL** sockets, you must conform to the expectations of the COMP system, which is a name in the format:

/comp/socket/0xNumber

The **/comp/socket/** part is an invariable prefix; the *0xNumber* is a string representation of a 16-bit number in hexadecimal format. The Wind River Network Stack does not allow any other format for the name.

Use this name when you **bind()** a name to a socket and when you **connect()** to a listening socket. Supply this name by using a **sockaddr_un** structure:

```
struct sockaddr_un          /* LOCAL (UNIX) family address */
{
    /* ----- */
    uint8_t  sun_len;        /* 0x00: structure size */
    uint8_t  sun_family;     /* 0x01: address family */
    char     sun_path [104]; /* 0x02: actual address */
};                          /* 0x6A: TOTAL SIZE */
```

This structure is defined in **un.h**. You will need to include **un.h** as well as **sockLib.h**, the include file for the standard socket API:

```
#include <sockLib.h> /* standard socket API */
#include <sys/un.h>  /* struct sockaddr_un definition */
```

The following example sets up a **sockaddr_un** structure for an **AF_LOCAL** socket:

```
mySockaddr_un.sun_len = sizeof (struct sockaddr_un); /* 106 bytes */
mySockaddr_un.sun_family = AF_LOCAL;
bcopy ("/comp/socket/0x1234", mySockaddr_un.sun_path, 20);
```

10.5 Working with Internet Domain Sockets

This section outlines how to set up an Internet socket, how to connect it to a peer, how to exchange data with that peer, and how to shut down the socket.

A socket is a communications endpoint to which you can bind a name. Under VxWorks, a socket is implemented and managed using a socket descriptor, to which you can bind a name from the desired domain. To create a socket descriptor, call **socket()**. To configure this socket descriptor for your particular use of that socket, call **setsockopt()**. To bind a name to a socket descriptor, call **bind()**.

Creating a Socket Descriptor

To send and receive data over a socket, you need a socket descriptor. To create a socket descriptor, call **socket()**. As input, a VxWorks **socket()** call expects you to specify three things:

- a communications domain for the socket
- a socket type
- a protocol type

For the communications domain, specify either **AF_INET** or **AF_INET6**, depending on whether you want a socket in the IPv4 or IPv6 domain. For the socket type, use **SOCK_STREAM** to indicate a stream type socket. To indicate a datagram type socket, use **SOCK_DGRAM**.

In some cases you should also specify a protocol type, but this is not always necessary. For instance, for a stream socket in an Internet domain, you could specify the protocol type of **IPPROTO_TCP** (TCP provides the necessary services for a streams-type socket in the Internet domains). And for datagram sockets, you could specify a protocol type of **IPPROTO_UDP** (UDP is the IP protocol that provides the necessary services for a datagram type socket in the Internet domain). However, in these cases you do not need to specify either protocol type explicitly. Instead, you can set the protocol type to 0 (zero), which tells **socket()** to choose whatever transport layer protocol is best suited to the requested socket type and communications domain.

For raw sockets in the Internet domains, a protocol of zero indicates the IP protocol, **IPPROTO_IP**. Such a socket gives you direct access to IP. If you want an Internet domain raw socket that uses a different protocol, use an appropriate **IPPROTO_name** value as defined in **in.h**.

For example, to verify that an IP address is not already in use, a DHCP server implementation might want to send out an ICMP echo request on the socket to test the address. To do this, it would first create a raw socket in the IPv4 Internet

domain and specify a protocol of `IPPROTO_ICMP` (or, in the IPv6 Internet domain, it would specify a protocol of `IPPROTO_ICMPV6`).

Setting Socket Options

After creating a socket descriptor, you may want to specify the socket options that configure the socket for your particular needs. You can set most options with a call to `setsockopt()`, although there is one “option” you set with an `ioctl()` call.

Using an `ioctl()` Call to Make the Socket Non-Blocking

To make a socket non-blocking, call `ioctl()` as follows:

```
on = 1;
if (ioctl (mySocketDescriptor, FIONBIO, (int) &on) == -1)
{
    your response to the ioctl call failure
}
```

Calling `setsockopt()` to Set Socket Options

The synopsis for `setsockopt()` is defined as follows:

```
STATUS setsockopt
(
    int     s,          /* target socket */
    int     level,      /* protocol level of option */
    int     optname,    /* option name */
    char *  optval,     /* pointer to option value */
    int     optlen      /* option length */
)
```

This routine sets the options associated with a socket and any underlying protocols that enable the services accessible through the socket. Some `setsockopt()` calls you make before calling `connect()` or `bind()`. This is because some socket options affect the outcome of the `bind()` or the way the connect operation is formed. For example, some socket options let you restrict the `bind()` to a particular range of port numbers.

Some of the defaults for socket options in the Wind River Network Stack are as follows:

Table 10-2 Default Values for Socket Options

Option		Default Value
<code>SO_LINGER</code>	off	
<code>SO_REUSEADDR</code>	disabled	

Table 10-2 **Default Values for Socket Options** (cont'd)

Option		Default Value
SO_KEEPAIVE	disabled	
SO_DONTROUTE	disabled	
SO_RCVLOWAT	1	
SO_SNDLOWAT	1	
SO_ACCEPTCONN	no default, returns 1 if listen() has been called on the socket, 0 otherwise	
SO_BROADCAST	enabled (Note: this is opposite to the default in Linux)	
SO_USELOOPBACK	enabled	
SO_SNDBUF	IPNET_SOCK_DEFAULT_SEND_BUFSIZE (see Table 10-1)	
SO_RCVBUF	IPNET_SOCK_DEFAULT_RECV_BUFSIZE (see Table 10-1)	
SO_RCVTIMEO	infinite	
SO_ERROR	no default, return and clear the last socket error	
SO_TYPE	no default, returns the second argument passed to socket()	
SO_BINDTODEVICE	not bound to a device	
SO_OOINLINE	off	



NOTE: You do not have to call **setsockopt()**. You can use sockets without setting any socket options if the default options associated with a stream, datagram, or routing socket are appropriate to your needs. However, it is a good idea to know the default socket option values for options such as **SO_LINGER** and **SO_REUSEADDR**, and their consequences for the behavior of the socket.

The **level** parameter of a **setsockopt()** call lets you specify whether the option applies to the socket layer or to an underlying protocol. The socket-level options (level is **SOL_SOCKET**) are the most generic options and make sense in the context of many sorts of sockets. Names (**optname** values) for these socket options are defined in **socket.h**. In addition to these socket-level options, **setsockopt()** also lets

you pass options through to the underlying protocol or protocols that enable and implement the services that you access through the socket.

For example, there are many options associated with IPv4. These options control and configure access to IPv4-supplied functionality such as IPv4-based multicasting. To set these, specify a **level** of **IPPROTO_IP**. Names (**optname** values) for these socket options are defined in **in.h**.

The **setsockopt()** reference entry describes the most commonly used of the IP socket options listed above. Paralleling the socket options for IPv4, the Wind River Network Stack also has IPv6 socket options. For IPv6, you would assign **level** a value of **IPPROTO_IPV6** and use the **optname** values for these socket options as defined in **in6.h**.

Another commonly socket-accessed protocol and option is the **IPPROTO_TCP** protocol and **TCP_NODELAY** option, which is described in the **setsockopt()** reference entry. Other socket option names for this protocol are defined in **tcp.h**. Some protocols allow for socket access and some do not.

Binding a Socket to a Local Address

To bind a socket descriptor to a local address (sometimes called a name), call **bind()**. As input, a VxWorks **bind()** call expects you to specify three things:

- the socket to which you would bind a name
- the name, in the form of a **sockaddr**-like structure
- the length of the name

As input, **bind()** expects the socket name to be supplied using a pointer to a **sockaddr** structure. The layout of the **sockaddr** structure is not convenient for setting up all the information needed to specify a socket name in the Internet communications domain. Instead, use either a **sockaddr_in** structure or a **sockaddr_in6** structure that you cast to a **sockaddr** structure when you make the actual **bind()** call. Use the **sockaddr_in** structure for sockets in the IPv4 Internet communications domain, and the **sockaddr_in6** structure for sockets in the IPv6 Internet communications domain.



NOTE: Sockets in the routing domain never require an explicit **bind()** call.

If you do not call **bind()** for the socket descriptor that you create, the first time you call **connect()** or **sendto()** for the socket, the system automatically binds for you. In such a case, the system chooses the port number for your socket from among the unused private port numbers (also known as *dynamic* or *ephemeral* port numbers), in the range of **AnonPortMin** to **AnonPortMax** (see [Table 10-1](#)). The stack chooses the local IP address appropriately to reach the specified destination. This

association of a local IP address with the socket is temporary in the case of a **sendto()** (and may change from packet to packet), but the system maintains its port number assignment until you close the socket. This approach is acceptable if, as is common in client applications, you do not need to know the number of the port to which your socket binds.

Letting the System Select Your Port Number

A **bind()** call on an Internet-domain socket passes a socket address containing both a local IP address and a local transport-layer port number. You may specify either or both of these as zero. If you specify a port number of zero, the system chooses an available (nonzero) local port number; you can obtain the port number later if necessary by using the **getsockname()** call. If you specify an IP address of zero, the socket endpoint will accept connections or datagrams destined to any of the host's local IP addresses (with the chosen destination port number).

Binding to Well-Known and Registered Ports

In the Internet communications domains, port numbers 0 through 1023 are reserved for use by well-known services associated with TCP. These are services such as echo and time. Ports 1024 through 49151 are reserved for services registered with IANA. If you implement a server for a well-known or registered service, bind your socket to the port assigned to that service, and then monitor that socket for requests from clients.

For a list of the port assignments for well-known and registered services, see:

<http://www.iana.org/assignments/port-numbers>

This list is also useful if you are implementing a client of a well-known service. It tells you the port on the remote site to which you send requests.

If you need to request a port number assignment for a new service, you can apply to IANA. You need do that only if you are creating a service that you will to publish to the entire Internet. For a service offered within a network throughout which you can control how port numbers are used, you can choose a private port number within the range 49152 through 65535.

Setting Up a `sockaddr_in` Structure to Store an IPv4 Address

The `in.h` file defines the `sockaddr_in` structure as follows:

```
struct sockaddr_in
{
    u_char          sin_len;          /* The length of the address.    */
    u_char          sin_family;       /* The address family, AF_INET.  */
    u_short         sin_port;         /* The port number.             */
    struct in_addr  sin_addr;         /* The IP Address.              */
    char            sin_zero[8];      /* Optional mask for address.    */
};

struct in_addr
{
    in_addr_t s_addr;
};
```

The `in_addr_t` is an unsigned 32-bit integer; its value is the IPv4 address stored in network byte order.



NOTE: You must assign a value to `sin_len`. If you do not do this before you call, for example, `sendmsg()`, such a call will fail with an `EINVAL` (0x22) error.

Example 10-1 Setting Up a `sockaddr_in` Structure to Store an IPv4 Address

The following code fragment shows one way to set up a `sockaddr_in` structure for a server identified by a user-entered text string. This string can contain either a dot-notation IPv4 address or its user-friendly equivalent name.

```
struct addrinfo    hints;
struct addrinfo *  res;
int               r;
struct sockaddr_in serverAddr;

#define STRINGIFY(expr) #expr

hints.ai_family = AF_INET;
r = getaddrinfo (serverName, STRINGIFY(SERVER_PORT_NUM), &hints,
                &result);
if (r != 0)
{
    fprintf (stderr, "%s\n", gai_strerror (r));
    return ERROR;
}
bcopy (result->ai_addr, &serverAddr, result->ai_addrlen);
freeaddrinfo (result);
```

The `getaddrinfo()` call creates a socket address structure from the `serverName` and the `SERVER_PORT_NUM`. You can specify the `serverName` either as a dot-notation IPv4 address or as a name. If you configured the stack to include DNS, `getaddrinfo()` uses DNS to get the name if the host table search fails.

Setting Up a `sockaddr_in6` Structure to Store an IPv6 Address

The `in6.h` file defines the `sockaddr_in6` structure as follows:

```
struct sockaddr_in6
{
    u_int8_t      sin6_len;    /* length of this struct(sa_family_t) */
    u_int8_t      sin6_family; /* AF_INET6 (sa_family_t) */
    u_int16_t     sin6_port;   /* Transport layer port # (in_port_t) */
    u_int32_t     sin6_flowinfo; /* IP6 flow information */
    struct in6_addr sin6_addr; /* IP6 address */
    u_int32_t     sin6_scope_id; /* scope zone index */
};

struct in6_addr
{
    union
    {
        u_int8_t  __u6_addr8[16];
        u_int16_t __u6_addr16[8];
        u_int32_t __u6_addr32[4];
    } __u6_addr;
};
```

Example 10-2 Setting Up a `sockaddr_in6` Structure to Store an IPv6 Address

Consider the following code fragment:

```
struct addrinfo  hints;
struct addrinfo * res;
int             r;

#define STRINGIFY(expr) #expr

bzero ((char*) &hints, sizeof (struct sockaddr_in6));
hints.ai_family = AF_INET6;
r = getaddrinfo (serverName, STRINGIFY(SERVER_PORT_NUM), &hints,
                &result);

if (r != 0)
{
    fprintf (stderr, "%s\n", gai_strerror (r));
    return ERROR;
}
bcopy (result->ai_addr, &serverAddr, result->ai_addrlen);
freeaddrinfo (result);
```

The `getaddrinfo()` call creates a socket address structure from the **serverName** and the **SERVER_PORT_NUM**. You can specify the **serverName** either as a string representation of an IPv6 address or as a name. If you configured the stack to include DNS, `getaddrinfo()` uses DNS to get the name if the host table search fails.

Example 10-3 Calling getaddrinfo() for IPv4 or IPv6

The following example code illustrates the use of **getaddrinfo()** in a TCP client to establish a connection to a server:

```
#include <vxWorks.h>
#include <string.h>
#include <errnoLib.h>
#include <netdb.h>
#include <netinet/in.h>
#include <sockLib.h>
#include <ioLib.h>

int connectToServer
(
    char * serverName, /* "fe80::1:0:34ea%fe10", "www.foo.com", etc. */
    char * serviceName /* "ftp", "http", "41834", etc. */
)
{
    struct addrinfo hints;
    struct addrinfo * firstres;
    struct addrinfo * res;
    int rc;
    int sock;
    int size;

    memset (&hints, 0, sizeof (hints));

    hints.ai_family = AF_UNSPEC; /* allow either IPv4 or IPv6 */
    hints.ai_protocol = IPPROTO_TCP; /* let's restrict to TCP */

    rc = getaddrinfo (serverName, serviceName, &hints, &firstres);
    if (rc != 0)
    {
#ifdef DEBUG
        int err = errno;
        printf ("getaddrinfo() failed, returned: %d (%s), errno=0x%x\n",
            rc, gai_strerror (rc), err);
#endif
        return ERROR;
    }

    /*
     * Simple strategy: Try the returned addresses until they run out,
     * or find one which works. Warning: if there are several addresses
     * returned, and none of them works, pause here a while...
     */

    res = firstres; /* There's guaranteed to be at least one ... */

    do
    {
        sock = socket (res->ai_family, res->ai_socktype,
            res->ai_protocol);
    }
```

```
    if (sock < 0)
        continue;

    /*
     * Set desired socket options. Set socket buffer sizes,
     * and treat (unlikely) setsockopt() failures as non-fatal.
     */

    size = 65536;
    setsockopt (sock, SOL_SOCKET, SO_SNDBUF, (char *) &size,
                sizeof (size));
    setsockopt (sock, SOL_SOCKET, SO_RCVBUF, (char *) &size,
                sizeof (size));

    if (connect (sock, res->ai_addr, res->ai_addrlen) == 0)
        break;

    close (sock); /* failed to connect, try with next address */
    sock = ERROR;

} while ((res = res->ai_next) != NULL);

freeaddrinfo (firstres); /* free all the addrinfo structures */

return sock;
}
```

The example above attempts to establish a client TCP connection to a service specified as a server host name (which may be either a numerical IPv4 or IPv6 address, or a partially- or fully-qualified domain name), and a service name (which may be a decimal numeric port number, or one of a few named services (such as "ftp", "telnet", or "http").

The return value from this **connectToServer()** routine is **ERROR** on failure, or a socket file descriptor if successful. You can use the returned socket descriptor to communicate with the server according to the specified application-layer protocol over TCP.

The routine uses **getaddrinfo()** to do the name/service look-up and to retrieve parameters such as the socket address, socket address length, socket domain, socket type, and socket protocol that it uses when it calls **socket()**, **bind()**, and **connect()**.

10.5.1 Creating the Connection for Internet Domain Stream Sockets

You send and receive data through a socket in more-or-less the same manner whether the socket is a stream or datagram socket, but before you can exchange data over a stream socket, you must establish a connection. The routines used to create this connection assume a client server relationship.

A client wishing to establish a connection with a server calls **connect()** on a socket it has created to communicate with its server. Whether the **connect()** call blocks if a connection is not immediately available depends on how you have configured the socket (see *Setting Socket Options*, p.157).

If the socket is non-blocking, **connect()** returns immediately. If **connect()** succeeds in establishing a connection, it returns **OK**. Otherwise, it returns **ERROR** and sets **errno** to **EINPROGRESS** or **EALREADY** which indicates that the system is still attempting to complete the connection. You may then repeat the **connect()** call until the system establishes a connection, at which time **connect()** will again return **ERROR**, but the **errno** value will be **EISCONN**.

On the server side, the server must have called **listen()** on the listening socket it has created to listen for incoming client connections. The server can then call **accept()** on the socket to wait for a new client connection.

When the **accept()** call returns, the return value is either **ERROR** or a new socket descriptor. The **accept()** call may return **ERROR** under normal operation if the peer resets the connection after it has completed but before it is accepted. In such a case, **errno** is set to **ECONNABORTED**. The server should treat this as a non-fatal error, and call **accept()** again to await further client connections. The server should treat **EINTR** and **EWOULDBLOCK** **errno** values the same way. Other **errno** values set by **accept()** indicate an error that is likely unrecoverable.

If **accept()** returns a socket descriptor rather than **ERROR**, the server can use this descriptor to exchange messages with the client that requested the connection. The server may choose to pass this socket descriptor to a new task while the server returns its attention to accepting connection requests on the original socket.

Servers typically call **accept()** from within a “forever” loop. If there is no connection request pending, the **accept()** blocks until a connection request arrives on the socket. If this does not meet your needs, you can configure the socket to be non-blocking (see *Setting Socket Options*, p.157) or you can use **select()**. If no clients are requesting a connection over the socket, and the socket is non-blocking, the server’s **accept()** call returns **ERROR** with **errno** set to **EWOULDBLOCK**.

Using a **connect()** Call with a Datagram Socket

Calling **connect()** for a datagram socket does not cause any actual communication with the specified destination. It does configure the socket to assume the specified destination for all messages read from and written to the socket. This makes it possible to call **send()** and **recv()** on the datagram socket instead of **sendto()** and **recvfrom()**. Connected datagram sockets will only accept received datagrams whose source matches the specified peer address in the connect call.

10.5.2 Sending and Receiving Data Using Internet Domain Sockets

The mechanics of sending and receiving data on a socket are simple. Forming or interpreting the message can be more complicated, and varies according to the application protocol assumed by the sender and recipient. For the most part, the mechanics of forming protocol-appropriate messages is beyond the scope of this document. An exception is the discussion of routing-socket messages provided in *Wind River Network Stack for VxWorks 6 Programmer's Guide, Volume 1: Transport and Network Protocols*.

Sending and Receiving Data without Regard to the Control Data

If you have called **connect()** on a socket descriptor, or have received the socket descriptor from an **accept()** call, you can exchange messages over the socket using **send()** and **recv()** or the generic **write()** and **read()** routines. All four of these routines are typically used with stream sockets. If you called **connect()** on a datagram socket, you can also use these routines to communicate over that socket.

For a datagram socket for which you have not called **connect()**, you need to explicitly identify the communications partner on the other side of the socket. In such a situation, you can use **sendto()** and **recvfrom()** to exchange messages over the socket.

Accessing and Setting the Control Data for a Socket Message

The **sendmsg()** and **recvmsg()** routines let you work with the entirety of the socket message—both its message header and its data payload. You can access the message header by using a **msghdr** structure, which is defined in **socket.h** as:

```
/*
 * Message header for recvmsg and sendmsg calls.
 * Used value-result for recvmsg, value only for sendmsg.
 */
struct msghdr
{
    void *      msg_name;          /* optional address */
    socklen_t   msg_namelen;       /* size of address */
    struct iovec * msg_iov;        /* scatter/gather array */
    int         msg_iovlen;        /* # elements in msg_iov */
    void *      msg_control;       /* ancillary data, see below */
    socklen_t   msg_controllen;    /* ancillary data buffer len */
    int         msg_flags;         /* flags on received message */
};
```

When you use the **sendmsg()** and **recvmsg()** routines with a datagram socket for which you have not called **connect()**, use the **msg_name** field to specify the address of the communication partner on the other side of the socket.

The advantage of using **sendmsg()** and **recvmsg()** is that you can access the control (ancillary) information associated with the message, and not merely the message payload data. The **msg_control** field points to a buffer containing the ancillary data. You can access the data objects in this control buffer using **cmsghdr** structures, which **socket.h** defines as:

```
/*
 * Header for ancillary data objects in msg_control buffer.
 * Used for additional information with/about a datagram
 * not expressible by flags. The format is a sequence
 * of message elements headed by cmsghdr structures.
 */
struct cmsghdr
{
    socklen_t  msg_len;      /* data byte count, including hdr */
    int        msg_level;    /* originating protocol */
    int        msg_type;     /* protocol-specific type */
    /* followed by u_char msg_data[]; */
};
```

To make it easier to work with these message headers, **socket.h** defines the macros **MSG_FIRSTHDR()**, **MSG_DATA()**, and **MSG_NXTHDR()** as follows:

```
/* given pointer to struct cmsghdr, return pointer to data */
#define MSG_DATA(msg) ((u_char *) (msg) + \
    _ALIGN(sizeof(struct cmsghdr)))

/* given pointer to struct cmsghdr, return pointer to next cmsghdr */
#define MSG_NXTHDR(mhdr, msg) \
    (((caddr_t) (msg) + _ALIGN((msg)->msg_len) + \
    _ALIGN(sizeof(struct cmsghdr)) > \
    (caddr_t) (mhdr)->msg_control + (mhdr)->msg_controllen) ? \
    (struct cmsghdr *) NULL : \
    (struct cmsghdr *) ((caddr_t) (msg) + _ALIGN((msg)->msg_len)))

#define MSG_FIRSTHDR(mhdr) ((struct cmsghdr *) (mhdr)->msg_control)
```

This level of access is convenient when implementing a service such as **ping()**. If you are merely exchanging data and not interested in the control data, you are probably better off using the **send()**/**sendto()**/**write()** and **recv()**/**recvfrom()**/**read()** routines.

10.5.3 Closing or Shutting Down an Internet Domain Socket Connection

When a client or server has decided that it is time to end its conversation with the communication partner, it can break the connection by calling **shutdown()** or its generic near equivalent, **close()**. Of the two routines, **shutdown()** provides the greater flexibility. Using **shutdown()**, you can break the send side of the socket connection, the receive side of the socket connection, or both.



WARNING: Do not close a socket descriptor in one task that is still in use by another task, as this is not safe in VxWorks (the same is true of all file descriptors). To wake up a task that is blocked waiting for an event on a socket, you may use **shutdown()** but not **close()**, as **shutdown()** leaves the socket descriptor valid. You must then use some application-specific synchronization mechanism to ensure that all application tasks are done with it before the socket is closed.

If you enabled the **SO_LINGER** socket option and set the **l_linger** member of the **struct linger** argument, there can be a degree of latency associated with closing a socket connection. Calling **close()** initiates the tear-down of the session (in TCP this adds a **FIN** segment to the transmission queue), but when **close()** returns, the session is not yet closed and will not be in the **CLOSED** state until tear-down negotiation is complete and the session is reset by one of the peers. The local pool will perform this reset if it no peer does it in the time specified by the **SO_LINGER** socket option.

By default, the **SO_LINGER** socket option is not enabled. To enable it, pass **setsockopt()** a **struct linger** argument with its **l_onoff** member set to 1, and its **l_linger** member set either to zero or to a timeout value specified in seconds. If **l_linger** is zero, the **close()** operation forces an abrupt close on the connection and sends a **RST** segment to the TCP peer. If **l_linger** specifies a nonzero timeout, the **close()** call will block until either the peer acknowledges all outstanding sent data and the local side's **FIN**, or until the specified **l_linger** timeout, or until TCP itself times out the connection (whichever occurs first). On the other hand, if you do not enable the **SO_LINGER** option for the socket, the **close()** call will return immediately, but TCP will initiate its graceful shutdown handshaking. If the local side did the close first, the connection will normally enter **TIME_WAIT** state on the local side and remain there for a timeout period of two times the maximum segment lifetime (MSL).

10.5.4 Support Routines for Working with Internet Addresses

To convert host names to their corresponding numerical IP addresses and back, the Wind River Network Stack provides **hostLib**, **inetLib**, and the following routines:

- **getaddrinfo()** – node name-to-address translation
- **getnameinfo()** – translate a socket address to a node name and service name
- **gethostbyname()** – name-to-address translation routine
- **gethostbyaddr()** – address-to-name translation routine

Unless you are concerned with image size, the best routines to use are **getaddrinfo()** and **getnameinfo()**, since they apply to both IPv4 and IPv6. Among the **hostLib** routines, **hostGetByName()** and **hostGetByAddr()** are IPv4-only.

Index

A

Abandoned state max time 54
accept() 142, 165
 blocking 149
Address Notify 149
AF_INET 151, 156
 specifying for communications domain 156
AF_INET6 151
AF_LOCAL 137, 143, 148, 151, 152, 155
AF_MOBILITY 151
AF_ROUTE 151, 152, 154
AF_TIPC 151, 153
all subnets local 75, 77
Allow bootp 56
Allow decline 56
Allow dynamic bootp 57
Allow Rapid Commit 109
allow-bootp 80
allow-dynamic-bootp 80
AnonPortMax 149, 159
AnonPortMin 149, 159
arp cache timeout 78
Authentication attempts before disconnect 26
Authentication callback routine 26
Authorized dhcp relay agent 57
auxp 138

B

Backup server IPv4 address 19
Backup server IPv6 address 19
bind() 142, 149, 155, 156
 parameters and use 159
boot file size 73, 74, 76
boot-file 80
Bootfile name 50
BOOTP 45
Bootstrap Protocol, *see* BOOTP
broadcast address 75, 77
 pinging 8
build configuration parameters, *see* configuration parameters

C

Check address 57
close() 167
CMMSG_DATA() 167
CMMSG_FIRSTHDR() 167
CMMSG_NXTHDR() 167
cmsghdr
 defined 167
command interpreter 6
communications domains, sockets 151
COMP 148, 154, 155

configuration parameters 46
connect() 142, 155, 159
 datagram sockets 165
 streams sockets 165
Connection-Oriented Message Passing, *see* COMP
connectWithTimeout() 142
cookie server(s) 73, 74, 76
Core 137

D

Data receive timeout 27
Data send timeout 27
DCHPV6 Authentication HMAC-MD5 Key 108
Debug logging facilities in ftpLib 25
Default lease time 53
default-lease-time 80
Default socket receive buffer size 150
Default socket send buffer size 150
default tcp time to live 78
DHCP
 client
 configuring and building 95
 options 94, 96, 101
 options, not pre-implemented 103
 overview 93
 shell commands 100
 overview
 relay agent
 API routines 91
 configuring and building 85
 configuring statically 87
 definition 46
 overview 83
 server 49
 API routines 71
 components 51
 configuration database 51
 configuring 52
 configuring dynamically 62
 configuring statically 59, 72
 configuring with shell commands 72
 DHCPs daemon 51
 dumping lease database at shutdown 70
 hook routines 66
 initial state 59
 lease database 52
 overview 50
 reading configuration values from a file 67
 restoring lease database at startup 67
 shell commands 62
DHCP Client 95
DHCP client port 96
DHCP offer time-out in milliseconds 97
DHCP Relay Agent 85
DHCP relay network pre-configuration 86
DHCP relay startup callback routine 86
DHCP Server 52
DHCP server network pre-configuration 57, 59
DHCP server port 96
DHCP server startup callback routine 58
DHCP server termination callback routine 58
DHCP Unique ID, *see* DUID
DHCP6 Client 120
DHCP6 Server 107
DHCP Client Interface Rebind List 131
DHCP_CLIENT_ID 96
DHCP_CLIENT_PORT 96
DHCP_DISCOVER_RETRIES 97
DHCP_IF_CLIENT_ID_LIST 99
DHCP_IF_REQ_OPTS_LIST 98
DHCP_INSTALL_CALLBACK_HOOK 100
DHCP_OFFER_TIMEOUT 97
DHCP_OPTION_CALLBACK_HOOK 100
DHCP_REQ_OPTS 96
DHCP_RFC2131_EXP_BACKOFF 97
DHCP_RFC2131_INIT_DELAY 96
DHCP_SERVER_PORT 96
dhcpc6 132
DHCP6_AUTHENTICATION_REALM 116, 121
DHCP6_DUID_EN_NUM 122
DHCP6_DUID_EN_VAL 122
DHCP6_DUID_IF 122
DHCP6_DUID_TYPE 122
DHCP6_HMAC_MD5_SECRET 121
DHCP6_IF_DNS_LIST 124
DHCP6_IF_ENUM_LIST 123
DHCP6_IF_HINTS_DEFAULT_ENUM_LIST

- 128
- DHCPC6_IF_HINTS_DEFAULT_PREFERRED_LIST 130
- DHCPC6_IF_HINTS_DEFAULT_PREFIX_LIST 129
- DHCPC6_IF_HINTS_DEFAULT_REBIND_LIST 131
- DHCPC6_IF_HINTS_DEFAULT_RENEW_LIST 131
- DHCPC6_IF_HINTS_DEFAULT_VALID_LIST 129
- DHCPC6_IF_IA_NA_DEFAULT_IAID_LIST 128
- DHCPC6_IF_IA_NA_DEFAULT_LIST 127
- DHCPC6_IF_INFO_REFRESH_DEFAULT_LIST 126
- DHCPC6_IF_INFO_REFRESH_LIST 125
- DHCPC6_IF_INFO_REFRESH_MAX_LIST 127
- DHCPC6_IF_INFO_REFRESH_MIN_LIST 126
- DHCPC6_IF_INFORMATION_ONLY_LIST 124
- DHCPC6_IF_RAPID_COMMIT_LIST 123
- DHCPC6_IF_SNTP_LIST 125
- DHCPDISCOVER 93
- dhcpr 88
 - DHCPR_CLIENT_PORT 85
 - DHCPR_HOPS_THRESHOLD 86
 - DHCPR_INSTALL_CALLBACK_HOOK 86
 - DHCPR_MAX_PKT_SIZE 85
 - DHCPR_NETCONF_SYSVAR 86
 - DHCPR_SERVER_PORT 85
 - DHCPR_START_CALLBACK_HOOK 86
 - DHCPREQUEST 97
- dhcps 62
 - config 65
 - host 63
 - interface 65
 - lease 64
 - option 64
 - pool 63
 - subnet 63
- DHCPS_ABANDONED_STATE_MAX_TIME 54
- DHCPS_ALLOW_BOOTP 56
- DHCPS_ALLOW_DECLINE 56
- DHCPS_ALLOW_DYNAMIC_BOOTP 57
- DHCPS_AUTHORIZED_AGENTS 57
- DHCPS_CLIENT_PORT 53
- DHCPS_DEFAULT_LEASE_TIME 53
- DHCPS_DO_ICMP_ADDRESS_CHECK 57
- DHCPS_EXPIRED_STATE_MAX_TIME 54
- DHCPS_INSTALL_CALLBACK_HOOK 58, 66
- DHCPS_LEASE_BOOTPC_MAX_TIME 55
- DHCPS_MAX_LEASE_TIME 53
- DHCPS_MIN_LEASE_TIME 53
- DHCPS_NETCONF_SYSVAR 57, 59
- DHCPS_OFFERED_STATE_MAX_TIME 55
- DHCPS_PKT_SIZE 56
- DHCPS_REBINDING_TIME 54
- DHCPS_RELEASED_STATE_MAX_TIME 55
- DHCPS_RENEWAL_TIME 54
- DHCPS_SERVER_PORT 53
- DHCPS_START_CALLBACK_HOOK 58, 66
- DHCPS_STOP_CALLBACK_HOOK 58, 66
- DHCPS6_ALLOW_RAPID_COMMIT 109
- DHCPS6_AUTHENTICATION_REALM 108
- DHCPS6_DUID_IFNAME 109
- DHCPS6_HMAC_MD5_SECRET 108
- DHCPS6_IF_RELAY_MAP_LIST 110
- DHCPS6_MAXHOP_COUNT 110
- DHCPS6_MODE 109
- DHCPv6 45
 - client
 - authentication keys 114
 - example 115
 - automatic mode 114
 - configuring and building 120
 - configuring dynamically 116
 - configuring statically 114
 - hints 128
 - options 119
 - options, user- or vendor-defined 119
 - overview 113
 - rapid commit 123
 - reconfigure messages 119
 - replay detection 118
 - shell commands 132
 - temporary address assignment 119
 - overview 46
 - relay agent
 - configuring and building 108
 - overview 107
 - server

- authentication keys 110
 - example 111
- configuring and building 108
- overview 107

DHCPV6 Authentication Realm 108

DNS

- clients 12
- component overview 12
- configuring and building 13
- domain name space 11
- iterative resolver lookups 13
- name servers 12
- recursive lookups 13
- resolver cache, flushing 16
- resolvers 12
- zone, setting the 15

DNS Client 13

DNS domain name 14

DNS primary name server 14

DNS quaternary name server 14

DNS secondary name server 14

DNS server listening port 15

DNS tertiary name server 14

DNSC_DOMAIN_NAME 14

DNSC_IP4_ZONE 15

DNSC_IP6_ZONE 15

DNSC_PRIMARY_NAME_SERVER 14

DNSC_QUATERNARY_NAME_SERVER 14

DNSC_RETRIES 15

DNSC_SECONDARY_NAME_SERVER 14

DNSC_SERVER_PORT 15

DNSC_TERTIARY_NAME_SERVER 14

DNSC_TIMEOUT 15

documentation 3

domain name 73, 74, 76

Domain Name Server (DNS), *see* DNS

domain name server(s) 72

domains, communications

- for sockets 151

don't fragment flag 8

DUID

- enterprise number 122
- enterprise value 122

DUID EN Number 122

DUID EN Value 122

DUID Interface 122

DUID Type 122

Dynamic Host Configuration Protocol, *see* DHCP, DHCPv6

E

EALREADY 165

ECONNABORTED 165

EINPROGRESS 165

EINTR 165

EINVAL 161

EISCONN 165

Enable proxy FTP support 27

ENETDOWN 149

envp 138

ethernet encapsulation 78

EWouldBlock 165

Expired state max time 54

extensions path 73, 74, 76

External Data Representation, *see* XDR

F

File Transfer Protocol, *see* FTP

finger server(s) 79

flags

- don't fragment 8

FTP

- see also* TFTP
- boot example 44
- client 23, 24
 - mode 30
 - using 29
 - verbosity 30
- configuring and building 24
- file permissions 42
- network devices, creating 43
- security 23
- server 23, 25
- transmission modes supported 24

FTP Client 24

- FTP Client Backend 24
- FTP initial directory 27
- ftp password 44
- FTP root directory 27
- FTP Server 24
- FTP timeout 25
- FTP transient fatal function 25
- FTP Transient response maximum retry limit 25
- FTP_DEBUG_OPTIONS 25
- FTP_TIMEOUT 25
- FTP_TRANSIENT_FATAL 25
- FTP_TRANSIENT_MAX_RETRY_COUNT 25
- FTP_TRANSIENT_RETRY_INTERVAL 25
- FTP6 Client Backend 24
- FTP6_REPLYTIMEOUT 25
- FTPS_AUTH_ATTEMPTS 26
- FTPS_AUTH_CALLBACK_HOOK 26
- FTPS_ENABLE_PROXY 27
- FTPS_INACTIVITY_TIMEOUT 29
- FTPS_INITIAL_DIR 27
- FTPS_INSTALL_CALLBACK_HOOK 26
- FTPS_LOCAL_PORT_BASE 28
- FTPS_MAX_SESSIONS 28
- FTPS_MODE 28
- FTPS_PEER_PORT_BASE 28
- FTPS_PORT_NUM 29
- FTPS_RECV_TIMEOUT 27
- FTPS_ROOT_DIR 27
- FTPS_SEND_TIMEOUT 27
- FTPS_SLEEP_TIME 29
- FTPS_SYS_TYPE 28

G

- getaddrinfo() 161, 168
 - example 163
- gethostbyaddr() 13, 168
- gethostbyname() 13, 168
- getnameinfo() 168
- getOptServ() 67
- getpeername() 143
- getsockname() 143, 160
- getsockopt() 143
- Global client identifier 96

- Global list of requested dhcp options 96

H

- Hop Count Limit 110
- host name 73, 74, 76
- hostAdd() 43
- hostGetByAddr() 169
- hostGetByName() 169
- hostLib 168
- hosts.allow 36
- hosts.equiv 36, 38

I

- IAID
- iam() 42
- IANA
 - DHCPv6 clients and 127
- IANA, port assignments 160
- ICMP 6
- ICMP6_ECHO_REPLY 10
- ICMP6_ECHO_REQUEST 10
- ICMPv6 10
- Identity Association for Non-temporary Addresses,
 - see IANA
- Identity Association ID, see *IAID*
- IEEE Std. 1003.1 146
- ifconfig 100
- impress server(s) 73, 74, 76
- in.h 147, 156, 159
- in_addr_t 161
- in6.h 147
- INCLUDE_FTP 24
- INCLUDE_FTP6 24
- INCLUDE_IPCOM_USE_AUTH 41
- INCLUDE_IPDHCP 95
- INCLUDE_IPDHCP6 120
- INCLUDE_IPDHCP6R 85
- INCLUDE_IPDHCP6S 52
- INCLUDE_IPDHCP6S6 107
- INCLUDE_IPDHCP6S6_CMD 107

INCLUDE_IPDNSC 13
INCLUDE_IPFTP_CMD 24
INCLUDE_IPFTPC 24
INCLUDE_IPFTPS 24
INCLUDE_IPNET_SOCKET 147, 148
INCLUDE_IPNET_USE_NETLINKSOCK 147, 148
INCLUDE_IPNET_USE_ROUTE SOCK 147, 148
INCLUDE_IPNET_USE_SOCK_COMPAT 147, 148
INCLUDE_IPNSLOOKUP_CMD 13
INCLUDE_IPPING_CMD 7
INCLUDE_IPPING6_CMD 7, 9
INCLUDE_IPSNTP_CMD 18
INCLUDE_IPSNTP_COMMON 18
INCLUDE_IPSNTPC 18, 21
INCLUDE_IPSNTPS 18
INCLUDE_IPTELNETS 39
INCLUDE_IPTFTP_CLIENT_CMD 32
INCLUDE_IPTFTP_COMMON 32, 33
INCLUDE_IPTFTPC 32
INCLUDE_IPTFTPS 32, 33
INCLUDE_NET_DRV 43
INCLUDE_NET_HOST_SETUP 38
INCLUDE_PING 7
INCLUDE_PING6 7
INCLUDE_RCP 37
INCLUDE_REMLIB 35
INCLUDE_RLOGIN 37, 38
INCLUDE_RPC 37
INCLUDE_RTP 149
INCLUDE_SC_SOCKLIB 147, 148
INCLUDE_SECURITY 37, 38, 39
INCLUDE_SOCKLIB 147, 148
INCLUDE_TELNET_CLIENT 39
INCLUDE_TFTP_CLIENT 32, 33
INCLUDE_UN_COMP 148, 155
INCLUDE_XDR 37
 RPC 37
INCLUDE_XDR_BOOL_T 37
inetLib 168
Install dhcp client callback routine 100
Install dhcp relay callback routine 86
Install dhcp server callback routines 58
Install ftp server callback routine 26
Interface Default Hints Status List 128, 129, 130, 131
Interface DNS Status List 124
Interface Hints Valid List 129, 130
Interface IA_NA Default List 127
Interface IAID List 128
Interface Information Only Status List 124
Interface Information Refresh Status List 125, 126
interface mtu 75, 77
Interface Name (DHCPv6 server configuration parameter) 109
Interface Preferred List 129, 130
Interface Prefix Hints List 129
Interface Rapid Commit Status List 123
Interface Relay Map List 110
Interface Renew List 131
Interface SNTP Status List 125
Interface specific list of client identifier 99
Interface specific list of requested dhcp options 98
Interface Status List 123
ioctl()
 setting socket options with 157
ip forwarding 73, 74, 76
ip time to live 75, 77
IP_IFF_UP 93
IP_IFF_X_DHCPRUNNING 93
IPCOM DHCP6 Server commands 107
IPCOM FTP client commands 24
IPCOM nslookup commands 13
IPCOM ping commands 7
IPCOM ping6 commands 7
IPCOM SNTP commands 18
IPCOM telnet port 41
IPCOM TFTP Commands 32
IPCOM_FILE_ROOT 27, 34
IPCOM_TELNET_AUTH_ENABLED 41
IPCOM_TELNET_PORT 41
ipd 100
ipdhcpc.client_identifier 96
ipdhcpc.client_port 96
ipdhcpc.discover_retries 97
ipdhcpc.interfaceName.client_identifier 99
ipdhcpc.interfaceName.requested_options 98
ipdhcpc.offer_timeout 97
ipdhcpc.requested_options 96
ipdhcpc.rfc2131_exponential_backoff 97

- ipdhcpc.rfc2131_init_delay 96
- ipdhcpc.server_port 96
- IPDHCPC_OPTCODE_ALL_SUBNETS_LOCAL 103
- IPDHCPC_OPTCODE_ARP_CACHE_TIMEOUT 103
- IPDHCPC_OPTCODE_BOOT_SIZE 103
- IPDHCPC_OPTCODE_BROADCAST_ADDRESS 103
- IPDHCPC_OPTCODE_CLIENT_IDENTIFIER 103
- IPDHCPC_OPTCODE_COOKIE_SERVERS 103
- IPDHCPC_OPTCODE_DEFAULT_IP_TTL 103
- IPDHCPC_OPTCODE_DEFAULT_TCP_TTL 103
- IPDHCPC_OPTCODE_DHCP_CLIENT_IDENTIFIER 95
- IPDHCPC_OPTCODE_DHCP_LEASE_TIME 94
- IPDHCPC_OPTCODE_DHCP_MAX_MESSAGE_SIZE 95
- IPDHCPC_OPTCODE_DHCP_MESSAGE 95
- IPDHCPC_OPTCODE_DHCP_MESSAGE_TYPE 95
- IPDHCPC_OPTCODE_DHCP_OPTION_OVERLOAD 103
- IPDHCPC_OPTCODE_DHCP_PARAMETER_REQUEST_LIST 95
- IPDHCPC_OPTCODE_DHCP_REBINDING_TIME 95
- IPDHCPC_OPTCODE_DHCP_RENEWAL_TIME 95
- IPDHCPC_OPTCODE_DHCP_REQUESTED_ADDRESS 103
- IPDHCPC_OPTCODE_DHCP_SERVER_IDENTIFIER 95
- IPDHCPC_OPTCODE_DOMAIN_NAME 94
- IPDHCPC_OPTCODE_DOMAIN_NAME_SERVERS 94
- IPDHCPC_OPTCODE_EXTENSIONS_PATH 103
- IPDHCPC_OPTCODE_FINGER_SERVERS 103
- IPDHCPC_OPTCODE_FONT_SERVERS 103
- IPDHCPC_OPTCODE_HOME_AGENTS 104
- IPDHCPC_OPTCODE_HOST_NAME 102, 104
- IPDHCPC_OPTCODE_IEEE802_3_ENCAPSULATION 104
- IPDHCPC_OPTCODE_IMPRESS_SERVERS 104
- IPDHCPC_OPTCODE_INTERFACE_MTU 104
- IPDHCPC_OPTCODE_IP_FORWARDING 104
- IPDHCPC_OPTCODE_IRC_SERVERS 104
- IPDHCPC_OPTCODE_LOG_SERVERS 104
- IPDHCPC_OPTCODE_LPR_SERVERS 104
- IPDHCPC_OPTCODE_MASK_SUPPLIER 104
- IPDHCPC_OPTCODE_MAX_DGRAM_REASSEMBLY 104
- IPDHCPC_OPTCODE_MERIT_DUMP 104
- IPDHCPC_OPTCODE_NAME_SERVERS 104
- IPDHCPC_OPTCODE_NETBIOS_DD_SERVER 104
- IPDHCPC_OPTCODE_NETBIOS_NAME_SERVERS 104
- IPDHCPC_OPTCODE_NETBIOS_NODE_TYPE 104
- IPDHCPC_OPTCODE_NETBIOS_SCOPE 104
- IPDHCPC_OPTCODE_NIS_DOMAIN 104
- IPDHCPC_OPTCODE_NIS_SERVERS 104
- IPDHCPC_OPTCODE_NON_LOCAL_SOURCE_ROUTING 104
- IPDHCPC_OPTCODE_NTP_SERVERS 94
- IPDHCPC_OPTCODE_PAD 94
- IPDHCPC_OPTCODE_PATH_MTU_AGING_TIMEOUT 104
- IPDHCPC_OPTCODE_PATH_MTU_PLATEAU_TABLE 104
- IPDHCPC_OPTCODE_PERFORM_MASK_DISCOVERY 104
- IPDHCPC_OPTCODE_POLICY_FILTER 105
- IPDHCPC_OPTCODE_RESOURCE_LOCATION_SERVERS 105
- IPDHCPC_OPTCODE_ROOT_PATH 105
- IPDHCPC_OPTCODE_ROUTER_DISCOVERY 105
- IPDHCPC_OPTCODE_ROUTER_SOLICITATION_ADDRESS 105
- IPDHCPC_OPTCODE_ROUTERS 94
- IPDHCPC_OPTCODE_STATIC_ROUTES 105
- IPDHCPC_OPTCODE_SUBNET_MASK 94
- IPDHCPC_OPTCODE_SWAP_SERVER 105
- IPDHCPC_OPTCODE_TCP_KEEPALIVE_GARBAGE 105
- IPDHCPC_OPTCODE_TCP_KEEPALIVE_INTERVAL 105
- IPDHCPC_OPTCODE_TIME_OFFSET 105

- IPDHCP_OPTCODE_TIME_SERVERS 105
- IPDHCP_OPTCODE_TRAILER_
- ENCAPSULATION 105
- IPDHCP_OPTCODE_VENDOR_CLASS_
- IDENTIFIER 105
- IPDHCP_OPTCODE_VENDOR_
- ENCAPSULATED_OPTIONS 105
- IPDHCP_OPTCODE_X_DISPLAY_MANAGER 105
- ipdhcpc_option_callback() 94, 100
 - example 102
 - implementing 101
- ipdhcpc6.duid.en.number 122
- ipdhcpc6.duid.if 122
- ipdhcpc6.duid.type 122
- ipdhcpc6.duid.value 122
- ipdhcpc6.if.enum 123
- ipdhcpc6.if.interfaceName.dns 124
- ipdhcpc6.if.interfaceName.ia_
 - na.default.hints.default.preferred 130
- ipdhcpc6.if.interfaceName.ia_
 - na.default.hints.default.prefix 129
- ipdhcpc6.if.interfaceName.ia_
 - na.default.hints.default.valid 129
- ipdhcpc6.if.interfaceName.ia_
 - na.default.hints.enum.default 128
- ipdhcpc6.if.interfaceName.ia_
 - na.default.hints.rebind 131
- ipdhcpc6.if.interfaceName.ia_
 - na.default.hints.renew 131
- ipdhcpc6.if.interfaceName.ia_na.default.iaid 128
- ipdhcpc6.if.interfaceName.ia_na.enum.default 127
- ipdhcpc6.if.interfaceName.information_only 124
- ipdhcpc6.if.interfaceName.information_refresh 125
- ipdhcpc6.if.interfaceName.information_
 - refresh.default 126
- ipdhcpc6.if.interfaceName.information_
 - refresh.maximum 127
- ipdhcpc6.if.interfaceName.information_
 - refresh.minimum 126
- ipdhcpc6.if.interfaceName.rapid_commit 123
- ipdhcpc6.if.interfaceName.sn timer 125
- ipdhcpc6_authdb_create_ll_duid() 114, 117
- ipdhcpc6_get_replay_counters() 118
- ipdhcpc6_set_auth_realm() 117
- ipdhcpc6_set_default_auth_key() 117
- ipdhcpc6_set_replay_counters() 118
- ipdhcpc6_user_authdb_config_finished() 116, 117
- ipdhcpr.ClientPort 85
- ipdhcpr.HopsThreshold 86
- ipdhcpr.PacketSize 85
- ipdhcpr.ServerPort 85
- ipdhcpr_config.c 87
- ipdhcpr_interface_status_set() 91
- ipdhcpr_netconf_sysvar 86, 87
 - commands in 87
- ipdhcpr_server_add() 91
- ipdhcpr_server_delete() 91
- ipdhcpr_start_hook() 86, 91
 - implementing 89
 - reading configuration values from a file with 89
- ipdhcps.allow_bootp 56
- ipdhcps.allow_decline 56
- ipdhcps.allow_dynamic_bootp 57
- ipdhcps.authorized_agents 57
- ipdhcps.client_port 53
- ipdhcps.default_lease_time 53
- ipdhcps.do_icmp_address_check 57
- ipdhcps.in_abandoned_state_max_time 54
- ipdhcps.in_bootp_state_max_time 55
- ipdhcps.in_expired_state_max_time 54
- ipdhcps.in_offered_state_max_time 55
- ipdhcps.in_release_state_max_time 55
- ipdhcps.max_lease_time 53
- ipdhcps.min_lease_time 53
- ipdhcps.packet_size 56
- ipdhcps.rebinding_time 54
- ipdhcps.renewal_time 54
- ipdhcps.server_port 53
- ipdhcps_cmd_dhcp() 67, 69
- IPDHCP_CONF_CODE_ALLOW_BOOTP 80
- IPDHCP_CONF_CODE_ALLOW_DYNAMIC_BOOTP 80
- IPDHCP_CONF_CODE_BOOT_FILE 80
- IPDHCP_CONF_CODE_LEASE_TIME_DFLT 80
- IPDHCP_CONF_CODE_LEASE_TIME_MAX 81
- IPDHCP_CONF_CODE_LEASE_TIME_MIN 81

- IPDHCPD_CONF_CODE_NEXT_SERVER_IP 81
- IPDHCPD_CONF_CODE_NEXT_SERVER_NAME 81
- IPDHCPD_CONF_CODE_REBINDING_TIME 81
- IPDHCPD_CONF_CODE_RENEWAL_TIME 81
- ipdhcps_config.c 57, 59, 72
- ipdhcps_config_option_reset() 71
- ipdhcps_config_option_set() 71
- ipdhcps_dhcp_option_add() 71
- ipdhcps_dhcp_option_delete() 71
- ipdhcps_host_add() 71
- ipdhcps_host_delete() 71
- ipdhcps_interface_status_set() 71
- ipdhcps_lease_db_dump() 52, 70, 71
- ipdhcps_lease_db_restore() 52, 71
- ipdhcps_netconf_sysvar 57, 59
 - commands available in 60
 - example 61
- IPDHCPD_OPTCODE_ALL_SUBNETS_LOCAL 75, 77
- IPDHCPD_OPTCODE_ARP_CACHE_TIMEOUT 78
- IPDHCPD_OPTCODE_BOOT_SIZE 73, 74, 76
- IPDHCPD_OPTCODE_BROADCAST_ADDRESS 75, 77
- IPDHCPD_OPTCODE_CLIENT_IDENTIFIER 79
- IPDHCPD_OPTCODE_COOKIE_SERVERS 73, 74, 76
- IPDHCPD_OPTCODE_DEFAULT_IP_TTL 75, 77
- IPDHCPD_OPTCODE_DEFAULT_TCP_TTL 78
- IPDHCPD_OPTCODE_DOMAIN_NAME 73, 74, 76
- IPDHCPD_OPTCODE_DOMAIN_NAME_SERVERS 72
- IPDHCPD_OPTCODE_EXTENSIONS_PATH 73, 74, 76
- IPDHCPD_OPTCODE_FINGER_SERVERS 79
- IPDHCPD_OPTCODE_FONT_SERVERS 78
- IPDHCPD_OPTCODE_HOME_AGENTS 79
- IPDHCPD_OPTCODE_HOST_NAME 73, 74, 76
- IPDHCPD_OPTCODE_IEEE802_3_ENCAPSULATION 78
- IPDHCPD_OPTCODE_IMPRESS_SERVERS 73, 74, 76
- IPDHCPD_OPTCODE_INTERFACE_MTU 75, 77
- IPDHCPD_OPTCODE_IP_FORWARDING 73, 74, 76
- IPDHCPD_OPTCODE_IRC_SERVERS 79
- IPDHCPD_OPTCODE_LOG_SERVERS 73, 74, 76
- IPDHCPD_OPTCODE_LPR_SERVERS 73, 74, 76
- IPDHCPD_OPTCODE_MASK_SUPPLIER 75, 77
- IPDHCPD_OPTCODE_MAX_DGRAM_REASSEMBLY 75, 77
- IPDHCPD_OPTCODE_MERIT_DUMP 73, 74, 76
- IPDHCPD_OPTCODE_NAME_SERVERS 72
- IPDHCPD_OPTCODE_NETBIOS_DD_SERVER 78
- IPDHCPD_OPTCODE_NETBIOS_NAME_SERVERS 78
- IPDHCPD_OPTCODE_NETBIOS_NODE_TYPE 78
- IPDHCPD_OPTCODE_NETBIOS_SCOPE 78
- IPDHCPD_OPTCODE_NIS_DOMAIN 78
- IPDHCPD_OPTCODE_NIS_SERVERS 78
- IPDHCPD_OPTCODE_NISPLUS_DOMAIN 79
- IPDHCPD_OPTCODE_NISPLUS_SERVERS 79
- IPDHCPD_OPTCODE_NNTP_SERVERS 79
- IPDHCPD_OPTCODE_NON_LOCAL_SOURCE_ROUTING 73, 74, 76
- IPDHCPD_OPTCODE_NTP_SERVERS 78
- IPDHCPD_OPTCODE_NWIP_DOMAIN 79
- IPDHCPD_OPTCODE_PATH_MTU_AGING_TIMEOUT 75, 77
- IPDHCPD_OPTCODE_PATH_MTU_PLATEAU_TABLE 75, 77
- IPDHCPD_OPTCODE_PERFORM_MASK_DISCOVERY 75, 77
- IPDHCPD_OPTCODE_POLICY_FILTER 75, 77
- IPDHCPD_OPTCODE_POP3_SERVERS 79
- IPDHCPD_OPTCODE_RESOURCE_LOCATION_SERVERS 73, 74, 76
- IPDHCPD_OPTCODE_ROOT_PATH 73, 74, 76
- IPDHCPD_OPTCODE_ROUTER_DISCOVERY 75, 77
- IPDHCPD_OPTCODE_ROUTER_SOLICITATION_ADDRESS 75, 77
- IPDHCPD_OPTCODE_ROUTERS 72
- IPDHCPD_OPTCODE_SMTP_SERVERS 79
- IPDHCPD_OPTCODE_STATIC_ROUTES 75, 77
- IPDHCPD_OPTCODE_STDA_SERVERS 80
- IPDHCPD_OPTCODE_STREETTALK_SERVERS

- 80
- IPDHCPs_OPTCODE_SUBNET_MASK 72
- IPDHCPs_OPTCODE_SWAP_SERVER 73, 74, 76
- IPDHCPs_OPTCODE_TCP_KEEPALIVE_
GARBAGE 78
- IPDHCPs_OPTCODE_TCP_KEEPALIVE_
INTERVAL 78
- IPDHCPs_OPTCODE_TIME_OFFSET 72
- IPDHCPs_OPTCODE_TIME_SERVERS 72
- IPDHCPs_OPTCODE_TRAILER_
ENCAPSULATION 75, 77
- IPDHCPs_OPTCODE_VENDOR_
ENCAPSULATED_OPTIONS 78
- IPDHCPs_OPTCODE_WWW_SERVERS 79
- IPDHCPs_OPTCODE_X_DISPLAY_MANAGER
79
- ipdhcps_pool_add() 71
- ipdhcps_pool_delete() 71
- ipdhcps_start_hook() 52, 58, 66
 - reading a configuration file with 62
- ipdhcps_stop_hook() 52, 58, 66, 70
- ipdhcps_subnet_add() 71
- ipdhcps_subnet_delete() 71
- ipdhcps6.authkey 108
- ipdhcps6.authrealm 108
- ipdhcps6.duid.ifname 109
- ipdhcps6.mode 109
- ipdhcps6.relay.hop_count_limit 110
- ipdhcps6.relay.map 110
- ipdhcps6.server.allow_rapid_commit 109
- ipdhcps6_authdb_add_client_authkey() 112
- ipdhcps6_authdb_add_server_authkey() 115, 117
- ipdhcps6_authdb_create_ll_duid() 111
- ipdhcps6_authdb_delete_client_authkey() 112
- ipdhcps6_authdb_server_no_auth() 116
- IPDNSc, *see* DNS
- ipdnsc.domainname 14
- ipdnsc.ip4.zone 15
- ipdnsc.ip6.zone 15
- ipdnsc.port 15
- ipdnsc.primaryrns 14
- ipdnsc.quaternaryrns 14
- ipdnsc.retries 15
- ipdnsc.secondaryrns 14
- ipdnsc.tertiaryrns 14
- ipdnsc.timeout 15
- ipdnsc_getipnodebyaddr() 12
- ipdnsc_getipnodebyname() 12
- ipftps.authentications 26
- ipftps.authsleep 29
- ipftps.dir 27
- ipftps.lportbase 28
- ipftps.max_sessions 28
- ipftps.port_number 29
- ipftps.pportbase 28
- ipftps.proxy 27
- ipftps.readonly 28
- ipftps.receive_timeout 27
- ipftps.root 27
- ipftps.send_timeout 27
- ipftps.session_timeout 29
- ipftps.system 28
- IPNET_SOCKET_ADDR_NOTIFY 149
- IPNET_SOCKET_ANON_PORT_MAX 149
- IPNET_SOCKET_ANON_PORT_MIN 149
- IPNET_SOCKET_DEFAULT_RECV_BUFSIZE 150,
158
- IPNET_SOCKET_DEFAULT_SEND_BUFSIZE 150,
158
- IPNET_SOCKET_MAX 150
- IPPROTO_ICMP 157
- IPPROTO_IP 156, 159
- IPPROTO_TCP 156, 159
- IPPROTO_UDP 156
- IPSNTP, *see* SNTP
- ipsntp.client.backup.addr 19
- ipsntp.client.backup.addr6 19
- ipsntp.client.multi.addr 20
- ipsntp.client.multi.addr6 20
- ipsntp.client.multi.if 20
- ipsntp.client.multi.if6 20
- ipsntp.client.poll.count 19
- ipsntp.client.poll.interval 20
- ipsntp.client.poll.timeout 21
- ipsntp.client.primary.addr 20
- ipsntp.client.primary.addr6 20
- ipsntp.server.mcast.addr 22
- ipsntp.server.mcast.addr6 22
- ipsntp.server.mcast.interval 22
- ipsntp.server.mcast.ttl 22

- ipsntp.server.precision 22
- ipsntp.server.stratum 23
- ipsntp.udp.port 21, 23
- ipsntp_config.h 21
- IPSNTP_USE_CLIENT 21
- IPSNTP_USE_SERVER 21
- iptftp.dir 34
- iptftp.retries 34
- iptftp.timeout 34
- IPtFTP, *see* TFTP
- irc server(s) 79

L

- l_linger 168
- l_onoff 168
- Lease for bootp client max time 55
- Lease rebinding time 54
- Lease renewal time 54
- level 159
- listen() 142, 158, 165
- local domain sockets, working with 154
 - including in an image 155
 - setting up 155
- Local port base number 28
- log server(s) 73, 74, 76
- LOGIN_PASSWORD 38, 40
- LOGIN_USER_NAME 38, 40
- lpr server(s) 73, 74, 76

M

- Managed address configuration flag 114
- max datagram reassembly 75, 77
- Max dhcp relay packet size 85
- Max number of simultaneous sessions 28
- Maximum lease time 53
- Maximum number of hops 86
- Maximum number of sockets 150
- max-lease-time 81
- merit dump path 73, 74, 76
- Minimum lease time 53

- min-lease-time 81
- mobile ip home agent 79
- Mode (DHCP server configuration parameter) 109
- msg_control 167
- msg_name 166
- msghdr
 - defined 166

N

- name server(s) 72
- name servers
 - addresses of 14
- netbios dgram distr server(s) 78
- netbios name server(s) 78
- netbios node type 78
- netbios scope 78
- netDevCreate() 42, 43
- netDrv 41
 - downloading run-time images 43
 - general usage information 43
- Netlink socket 147, 148
- NetWare/IP Domain Name 50
- Netware/IP Information 50
- network application protocols 5
 - adding to builds 6
 - configuring dynamically 6
- Network Sockets 147
- Network Time Protocol (NTP), *see* SNTP
- Network Virtual Terminal, *see* NVT
- next-server-ip 81
- next-server-name 81
- nis domain 78
- nis server(s) 78
- nis+ domain 79
- nis+ server(s) 79
- nntp server(s) 79
- non local source routing 73, 74, 76
- nslookup 13, 16
 - command-line options 16
- ntp server(s) 78
- NUM_FILES 150
- Number of DHCP client retries 97
- Number of retransmissions 19

Number of retries for DNS queries 15
NVT

O

Offered state max time 55
open() 43
Option callback routine 100
OPTION_RECONF_ACCEPT 119
OPTION_RECONF_MSG 119
OPTION_USER_CLASS 119
OPTION_VENDOR_CLASS 119
OPTION_VENDOR_OPTS 119
optname 158
Other stateful configuration flag 114

P

Packet size 56
password protection for Telnet and FTP,
 component 38, 39
path mtu aging timeout 75, 77
path mtu plateau table 75, 77
Peer port base number 28
perform mask discovery 75, 77
perform router discovery 75, 77
ping 6
 RTP example 136
 TTL field 9
PING client 7
ping.vxe 136
ping6 6, 9
PING6 client 7
policy filter 75, 77
pop3 server(s) 79
port numbers
 ephemeral 160
 registered 160
 well-known 160
Primary server IPv4 address 20
Primary server IPv6 address 20

R

rcmd() 35
rcmd_af() 35
read() 166
Read/write mode 28
Real Time Process, *see* RTP
rebinding-time 81
recv() 143, 166
recvfrom() 143, 166
recvmsg() 143, 166
registered port numbers 160
Released state max time 55
remLib 35
 associated configuration component 35
remLibInit() 35
Remote Command 35
remote file access
 netDrv, using 41
 permissions 42
 remLib configuration component 35
Remote Procedure Call, *see* RPC
remote shell, *see* RSH
renewal-time 81
Reported system type 28
resource location server(s) 73, 74, 76
RFC 854 39
RFC 951 45, 84
RFC 959 24
RFC 1014 37
RFC 1034 12
RFC 1035 12
RFC 1123 24
RFC 1350 31
RFC 1542 45, 84
RFC 1831 36
RFC 2030 17
RFC 2131 45, 49, 94
RFC 2132 45, 49, 60, 94
RFC 2242 50
RFC 2428 24
RFC 2462 129, 130
RFC 2463 10
RFC 2577 24
RFC 3315 46, 108, 118

- RFC 3493 146
- RFC 3542 146
- RFC 3596 12
- RFC 3646 119
- RFC 3736 119
- RFC 4075 119
- RFC 4242 119
- RFC2131 Exponential Back-off Delay 97
- RFC2131 Initialization Delay identifier 96
- .rhosts 36, 38, 41
- RLOGIN 37
- rlogin 37
 - security 37
- rlogin() 38
- rlogin/telnet encrypted password 38, 40
- rlogin/telnet user name 38, 40
- rlogLib 38
- root path 73, 74, 76
- route table
 - specifying 8, 10
- router solicitation address 75, 77
- router(s) 72
- routing socket support 147, 148
- RPC
- rpcLib 36
- rpcTaskInit() 36
- RSH 41
 - configuring and building 35
 - file permissions 42
 - network devices, creating 43
- RSH_STDERR_SETUP_TIMEOUT 35
- rshd 34, 41
- RTP
 - building 137
 - core network stack functionality and 136, 137
 - "Hello World" example 138
 - initialization functions 140
 - launching 139
 - network application as 136
 - overview 135
 - ping example 136
 - shutting down 141
 - socket calls in 136
 - socket connections 142
 - supported network applications 136

- RTP_ID 141
- rtpDelete() 141
- rtpKill() 142
- rtpShow() 142
- rtpSp() 136, 139
- rtpSpawn() 139

S

- Seconds between retransmissions 21
- send() 142, 166
- sendmsg() 142, 166
- sendto() 142, 159, 160, 166
- Server port number 29
- setsockopt() 143, 156, 157, 158
- shell commands 6
- shutdown() 143, 167
- Simple Network Time Protocol, *see* SNTP
- sin_len 161
- smtp server(s) 79
- SNTP 17
 - client 17
 - configuring 19
 - configuring and building 18
 - DHCPv6 clients and 125
 - server 17
 - server, configuring 22
 - server, enabling the 21
 - shell command 17
- SNTP Client 18
- SNTP common configurations 18
- SNTP multicast client mode interface 20
- SNTP multicast client mode IPv6 interface 20
- SNTP multicast client mode IPv6 multicast group 20
- SNTP multicast client mode multicast group 20
- SNTP multicast mode IPv4 destination address 22
- SNTP multicast mode IPv6 destination address 22
- SNTP multicast mode send interval 22
- SNTP multicast mode TTL 22
- SNTP port 21, 23
- SNTP Server 18
- SNTP server precision 22
- SNTP server stratum 23

- SNTP unicast client mode poll interval 20
- SNTP_LISTENING_PORT 21, 23
- SNTPC_BACKUP_IPV4_ADDR 19
- SNTPC_BACKUP_IPV6_ADDR 19
- SNTPC_MULTICAST_GROUP_ADDR 20
- SNTPC_MULTICAST_GROUP_IPV6_ADDR 20
- SNTPC_MULTICAST_MODE_IF 20
- SNTPC_MULTICAST_MODE_IPV6_IF 20
- SNTPC_POLL_COUNT 19
- SNTPC_POLL_INTERVAL 20
- SNTPC_POLL_TIMEOUT 21
- SNTPC_PRIMARY_IPV4_ADDR 20
- SNTPC_PRIMARY_IPV6_ADDR 20
- SNTPS_IPV4_MULTICAST_ADDR 22
- SNTPS_IPV6_MULTICAST_ADDR 22
- SNTPS_MULTICAST_INTERVAL 22
- SNTPS_MULTICAST_TTL 22
- SNTPS_PRECISION 22
- SNTPS_STRATUM 23
- SO_ACCEPTCONN 158
- SO_BINDTODEVICE 158
- SO_BROADCAST 158
- SO_DONTROUTE 158
- SO_ERROR 158
- SO_KEEPALIVE 158
- SO_LINGER 157, 158, 168
- SO_OOBLINKE 158
- SO_RCVBUF 158
- SO_RCVLOWAT 158
- SO_RCVTIMEO 158
- SO_REUSEADDR 157, 158
- SO_SNDBUF 158
- SO_SNDLOWAT 158
- SO_TYPE 158
- SO_USELOOPBACK 158
- SOCK_DGRAM 153, 156
- SOCK_RAW 153, 154
- SOCK_RDM 153
- SOCK_SEQPACKET 143, 148, 153, 154
- SOCK_STREAM 153, 156
- sockaddr 152, 159
- sockaddr_in 159
 - defined 161
 - example 161
 - setup of 161
- sockaddr_in6 152, 159
 - defined 162
 - example 162
 - setup of 162
- sockaddr_un 155
- Socket API 147, 148
- Socket API System Call support 147, 148
- Socket backend 147, 148
- socket descriptor
 - creating 156
- Socket support 147, 148
- socket() 142, 152, 156, 158
- socket.h 147, 158
- sockets
 - accepting connections 165
 - background reading 146
 - binding a name to 159
 - BSD socket compatibility 146
 - closing 167
 - communications domains 151
 - configuration components for 147
 - configuring and building 147
 - control information, accessing 166
 - creating 156
 - datagram
 - defined 153
 - file descriptors and 146
 - generic definition of 150
 - implicit 160
 - internet domain 151
 - inter-task communication with 143
 - IPv4 addresses, and 161
 - IPv6 addresses, and 162
 - listening for connections 165
 - local 152
 - macros accessing control information 167
 - non-blocking 157
 - options, setting 157
 - port numbers, ephemeral 160
 - raw 153
 - routing 152
 - sending/receiving data 166
 - sockaddr_in, setup of 161
 - sockaddr_in6, setup of 162
 - stream

- defined 153
- types 153
- VxWorks-specific issues 146
- sockLib 148, 152
- sockLib.h 155
- sockScLib 149
- SOL_SOCKET 158
- static routes 75, 77
- stda server(s) 80
- street talk server(s) 80
- subnet mask 72
- subnet mask supplier 75, 77
- swap server(s) 73, 74, 76
- sysctl() 137
- sysvar 6
- sysvars 46

T

- taskCreate() 140
- taskExit() 140, 141
- taskSpawn() 140
- taskSuspend() 141
- tcp keep alive garbage 78
- tcp keep alive interval 78
- tcp.h 147
- TCP_NODELAY 159
- telnet 39
 - client 39
 - configuring and building 39
 - security 39
 - server 40
- Telnet authentication 41
- TELNET client 39
- Telnet Server 39
- TELNET/FTP password protection 37, 39
- telnetcLib 39
- TFTP
 - configuring and building 32
 - using 32
- TFTP Client 32
- TFTP Client APIs 32
- TFTP Common Configurations 32
- TFTP number of retries 34

- TFTP retransmit timeout in seconds 34
- TFTP Server 32
- TFTP server (DHCP option) 50
- TFTP server working directory 34
- tftpLib
 - configuration component for 33
- TFTPS_DIRS 34
- TFTPS_RETRANSMIT_TIMEOUT 34
- TFTPS_RETRIES 34
- Time delay between retries after FTP_TRANSIENT encountered 25
- time offset 72
- time server(s) 72
- Time to sleep after authentication fail 29
- Timeout in seconds when waiting for responses to DNS queries 15
- Timeout interval for second RSH connection if any 35
- trailer encapsulation 75, 77
- Trivial File Transfer Protocol, *see* TFTP
- TTL field (ping) 9

U

- UDP port used by the DHCP server 53
- UDP port used by the dhcp server 85
- UDP port used by the dhcp/bootp clients 53, 85
- un.h 155
- User inactivity timeout 29
- usrAppInit() 116
- usrNetInit() 43

V

- Vendor class identifier 50
- vendor encapsulated options 78

W

- WAIT_FOREVER 35
- well-known port numbers 160

`write()` [166](#)
`_WRS_CONSTRUCTOR` [140, 141](#)
`www server(s)` [79](#)

X

X display manager [79](#)
X font server(s) [78](#)
XDR [37](#)
 RPC and [37](#)
XDR boolean support [37](#)

Z

Zone for IPv4 address to name lookups [15](#)
Zone for IPv6 address to name lookups [15](#)