DHCP Extension Dictionary

This appendix describes the DHCP extension dictionary entries and the application program interface (API) to the extension dictionary. It describes the data items available in the request and response dictionaries, and the calls to use when accessing dictionaries from Tcl extensions and shared libraries.

The appendix contains the following sections;

- Extension Dictionary Entries, on page 1
- Extension Dictionary API, on page 33
- Handling Objects and Options, on page 50
- Examples of Option and Object Method Calls, on page 52

Extension Dictionary Entries

A dictionary is a data structure that contains key-value pairs. There are two types of dictionaries: the attribute dictionaries that the request and response dictionaries use, and the environment dictionary. This section describes the request and response dictionaries; the environment dictionary entries are described in Tcl Environment Dictionary Methods, on page 37.

Decoded DHCP Packet Data Items

The decoded DHCPv4 packet data items represent the information in the DHCP packet, and are available in both the request and response dictionaries. These dictionaries provide access to considerably more internal server data structures than just the decoded request and decoded response.

All of the options followed by an asterisk (*) are multiple, which means that there can be more than one value associated with each option. In the DHCP/BOOTP packet, all of these data items appear in the same option. However, in the extension interface, these multiple data items are accessible through indexing.

You can access options that do not have names in Table 3: DHCPv4 and BOOTP Options , on page 2 as option–*n* , where *n* is the option number. All fields are read/write. Table 1: DHCPv4 and BOOTP Fields , on page 1 describes the field values for the DHCPv4 packets; Table 2: DHCPv6 Fields , on page 2 describes the field values for the DHCPv6 messages.

Table 1: DHCPv4 and BOOTP Fields

Name	Value
chaddr	blob (sequence of bytes)

Name	Value
ciaddr	IP address
file	string
flags	16-bit unsigned integer
giaddr	IP address
hlen	8-bit unsigned integer
hops	8-bit unsigned integer
htype	8-bit unsigned integer
op	8-bit unsigned integer
secs	16-bit unsigned integer
siaddr	IP address
sname	string
xid	32-bit unsigned integer
yiaddr	IP address

Table 2: DHCPv6 Fields

Name	Value
hop-count	8-bit unsigned integer
link-address	IPv6 address
msg-type	8-bit unsigned integer
peer-address	IPv6 address
xid	32-bit unsigned integer

The table below lists the DHCP and BOOTP options for DHCPv4.

Table 3: DHCPv4 and BOOTP Options

Name (*=multivalue)	Number	Value
all-subnets-local	27	byte-valued boolean
authentication	90	blob (sequence of bytes); 5 fields
auto-configure	116	8-bit unsigned integer
arp-cache-timeout	35	unsigned time

Name (*=multivalue)	Number	Value
bcmcs-servers-a*	89	IP address
bcmcs-servers-d*	88	DNS name
boot-file	67	string
boot-size	13	16-bit unsigned integer
broadcast-address	28	IP address
cablelabs-125(v-i-vendor-info ID: 4491)	125 suboption:	binary
oro	1	Option request, 8-bit unsigned integer (8-bit unsigned integers)
erouter-container	3	Erouter container options (binary; TLV encoded options)
packetcable-mib-env	4	MIB environment indicator (8-bit enumeration)
modem-capabilities	5	Modem capabilities encoding (binary; TLV5 encoded data)
dhcpv6-servers	123	DHCPv6 server suboptions (binary)
ip-pref	124	IPv4 or IPv6 preference (8-bit enumeration)
cablelabs-client- configuration	122 suboption:	blob (sequence of bytes)
[ccc-]primary-dhcp- server	1	IP address
[ccc-]secondary-dhcp- server	2	IP address
[ccc-]provisioning- server	3	blob (the first byte must be the type byte, with 0 for RFC 1035 encoding, and 1 for IP address encoding, for which the address must be in network order)
[ccc-]as-backoff-retry- blob	4	12-byte blob (3 unsigned 4-byte integers, which must be in network order); configures the Kerberos AS-REQ/AS-REP timeout, back-off, and retry mechanism
[ccc-]ap-backoff-retry- blob	5	12-byte blob (3 unsigned 4-byte integers, which must be in network order); configures the Kerberos AP-REQ/AP-REP timeout, back-off, and retry mechanism
[ccc-]kerberos-realm	6	variable-length blob (an RFC 1035 style name); a Kerberos realm name is required

Name (*=multivalue)	Number	Value
[ccc-]use-tgt	7	1-byte unsigned integer boolean; indicates whether to use a Ticket Granting Ticket (TGT) when obtaining a service ticket for one of the application servers
[ccc-]provisioning-timer	8	1-byte unsigned integer; defines the maximum time allowed for the provisioning process to finish
[ccc-]ticket-control- mask	9	2-byte unsigned integer, in host order
[ccc-]kdc-addresses- blob	10	variable-length (multiple of 4) IP address, in network order
cisco-autoconfigure	251	bounded byte
cisco-client-last-transaction- time	163	unsigned 32-bit integer
cisco-client-requested-host- name	162	string
cisco-leased-ip	161	IP address
cisco-vpn-id	221	blob (structured)
classless-static-route	121	blob (structured)
client-fqdn	81	blob (sequence of bytes); 4 fields: flags, rcode-1, rcode-2, and domain-name
cookie-servers*	8	IP address
default-ip-ttl	23	8-bit unsigned integer
default-tcp-ttl	37	8-bit unsigned integer
dhcp-class-identifier	60	string
dhcp-client-identifier	61	blob (sequence of bytes)
dhcp-lease-time	51	unsigned time
dhcp-max-message-size	57	16-bit unsigned integer
dhcp-message	56	string
dhcp-message-type	53	8-bit unsigned integer
dhcp-option-overload	52	8-bit unsigned integer
dhcp-parameter-request-list*	55	8-bit unsigned integer
dhcp-parameter-request- list-blob*	55	blob (sequence of bytes)
dhcp-rebinding-time	59	unsigned time

Name (*=multivalue)	Number	Value
dhcp-renewal-time	58	unsigned time
dhcp-requested-address	50	IP address
dhcp-server-identifier	54	IP address
dhcp-user-class-id	77	set of counted len byte arrays; 2 fields: typcnt-size and user-data
domain-name	15	string
domain-name-servers*	6	IP address
domain-search	119	blob (sequence of bytes)
extensions-path	18	string
finger-servers*	73	IP address
font-servers*	48	IP address
geo-conf	123	blob (sequence of bytes)
geoconf-civic	99	blob (sequence of bytes)
host-name	12	string
ieee802.3-encapsulation	36	byte-valued boolean
impress-servers*	10	IP address
initial-url	114	string
interface-mtu	26	16-bit unsigned integer
ip-forwarding	19	byte-valued boolean
irc-servers*	74	IP address
iSNS	83	blob (sequence of bytes); 7 fields
ldap-url	95	string
log-servers*	7	IP address
lost-server	137	DNS Name (see RFC 5223)
lpr-servers*	9	IP address
lq-associated-ip*	92	IP address
lq-client-last-transaction- time	91	unsigned time
mask-supplier	30	byte-valued boolean

Name (*=multivalue)	Number	Value
max-dgram-reassembly	22	16-bit unsigned integer
mcns-security-server	128	IP address
merit-dump	14	string
mobile-ip-home-agents*	68	IP address
name-servers*	5	IP address
name-service-search*	117	16-bit unsigned integer
nds-servers*	85	IP address
nds-tree	86	string
nds-context	87	string
netbios-dd-servers*	45	IP address
netbios-name-servers*	44	IP address
netbios-node-type	46	8-bit unsigned integer
netbios-scope	47	string
netinfo-parent-server-addr	112	IP address
netinfo-parent-server-tag	113	string
netwareip-domain	62	string
netwareip-information	63	blob (sequence of bytes)
nis+-servers*	65	IP address
nis+domain	64	string
nis-domain	40	string
nis-servers*	41	IP address
nntp-servers*	71	IP address
non-local-source-routing	20	byte-valued boolean
ntp-servers*	42	IP address
pana agent	136	IP address(es) (see RFC 5192)
path-mtu-aging-timeout	24	unsigned time
path-mtu-plateau-tables*	25	16-bit unsigned integer
perform-mask-discovery	29	byte-valued boolean

Name (*=multivalue)	Number	Value
policy-filters*	21	IP address (there can be two policy filters, each one having its own IP address)
pop3-servers*	70	IP address
posix-timezone	100	string (see RFC 4833)
pxe-client-arch	93	16-bit unsigned integer
pxe-client-machine-id	97	blob (sequence of bytes); 2 fields: type-flag and uuid
pxe-client-network-id	94	blob (sequence of bytes); 2 fields: type-flag and version
rapid-commit	80	null-length
relay-agent-info suboptions:	82 suboption:	blob (sequence of bytes)
relay-agent-circuit-id- data	1	blob (sequence of bytes)
relay-agent-remote-id- data	2	blob (sequence of bytes)
relay-agent-device- class-data	4	4-byte unsigned integer
relay-agent-subnet- selection-data	5	IP address
subscriber-id	6	string identifying the network client or subscriber
radius-attributes	7	supported attributes are user, class, and framed-pool
authentication	8	binary
v-i-vendor-opts	9	vendor options
cisco-subnet-selection	150	IP address
cisco-vpn-id	151	binary
cisco-server-id-override	152	IP address
relay-agent-vpn-id-data	181	string
relay-agent-server-id- override-data	182	IP address

Name (*=multivalue)	Number	Value
Note The relay-agent-circuit-id, relay-agent-remote-id, and relay-agent-device-class suboptions, which returned the two bytes (suboption code and data length) preceding the suboption data, are deprecated, but still available.		
resource-location-servers*	11	IP address
root-path	17	string
router-discovery	31	byte-valued boolean
router-solicitation-address	32	IP address
routers*	3	IP address
sip-servers	120	blob (sequence of bytes); 2 fields: flag and sip-server-list
slp-directory-agent*	78	blob (sequence of bytes); 2 fields: mandatory and agent-ip-list
slp-service-scope*	79	blob (sequence of bytes); 2 fields: mandatory and slp-scope-list
smtp-servers*	69	IP address
static-routes*	33	IP address
streettalk-directory-assistance-servers*	76	IP address
streettalk-servers*	75	IP address
subnet-alloc	220	blob (sequence of bytes); 5 fields: flags, subnet-request, subnet-info, subnet-name, and subnet-suggested-lease-time
subnet-mask	1	IP address
subnet-selection	118	IP address
swap-server	16	IP address
tcp-keepalive-internal	38	unsigned time

Name (*=multivalue)	Number	Value
tcp-keepalive-garbage	39	byte-valued boolean
time-offset	2	signed time
time-servers*	4	IP address
trailer-encapsulation	34	byte-valued boolean
tzdb-timezone	101	string (see RFC 4833)
user-auth	98	string
vendor-encapsulated-options	43	blob (sequence of bytes)
v-i-vendor-class	124	blob (sequence of bytes)
v-i-vendor-info	125	blob (sequence of bytes)
vpn-id	185	blob (structured); 2 fields: flag and vpn-id
www-servers*	72	IP address
x-display-managers*	49	IP address

The table below lists the DHCPv6 options.

Note Access to these options is available using the **putOption**, **getOption**, and **removeOption** methods only.

Table 4: DHCPv6 Options

Name (*=multivalue)	Number	Value
auth	11	binary; 5 fields: protocol, algorithm, replay-detection-method, replay-detection, and auth-info
bcmcs-server-a*	34	IPv6 address
bcmcs-server-d*	33	DNS name
cablelabs-17(vendor-opts ID: 4491)	17 suboption:	vendor-opts; 20 suboptions
oro	1	16-bit unsigned integer
device-type	2	string
embedded-components-list	3	string
device-serial-number	4	string
hardware-version-number	5	string

Name (*=multivalue)	Number	Value
software-version-number	6	string
boot-rom-version	7	string
vendor-oui	8	string
model-number	9	string
vendor-name	10	string
ecm-cfg-encaps	15	string
config-file-name	33	string
syslog-servers	34	IPv6 address
modem-capabilities	35	binary
device-id	36	binary
rfc868-servers	37	IPv6 address
time-offset	38	unsigned time
ip-pref	39	8-bit unsigned integer
cmts-capabilities	1025	binary; 1 suboption: docsis-version
cm-mac-address	1026	binary
erouter-container	1027	binary
cablelabs-client-configuration	2170 suboption:	IPv6 address; 2 suboptions (various data types)
primary-dhcp-server	1	IP address
secondary-dhcp-server	2	IP address
cablelabs-client-configuration-v6	2171 suboption:	IPv6 address; 9 suboptions (various data types)
primary-dhcpv6-server- selector-id	1	binary
secondary-dhcpv6-server- selector-id	2	binary
provisioning-server	3	binary
as-backoff-retry	4	binary
ap-backoff-retry	5	binary
kerberos-realm	6	DNS name

Name (*=multivalue)	Number	Value
use-tgt	7	unsigned 8-bit
provisioning-timer	8	unsigned 8-bit
ticket-control-mask	9	unsigned 16-bit
client-data	45	binary (options) (see RFC 5007)
client-fqdn	39	binary; 2 fields: flags and domain-name
client-identifier	1	binary
clt-time	46	32-bit unsigned time (see RFC 5007)
dns-servers*	23	IPv6 address
domain-list*	24	DNS name
elapsed-time	8	unsigned 16-bit
ero	43	unsigned 16-bit (see RFC 4994)
geoconf-civic	36	binary
ia-na	3	binary; 3 fields: iaid, t1, and t2
ia-pd	25	binary; 3 fields: iaid, t1, and t2
ia-prefix	26	binary; 4 fields: preferred-lifetime, valid-lifetime, prefix-length, and prefix
ia-ta	4	binary; 1 suboption: iaid
iaaddr	5	binary; 3 fields: address, preferred-lifetime, and valid-lifetime
info-refresh-time	32	unsigned time
interface-id	18	binary
lost-server	51	DNS Name (see RFC 5223)
lq-client-links	48	IPv6 address(es) (see RFC 5007)
lq-query	44	binary structured (see RFC 5007)
lq-relay-data	47	binary (DHCPv6 message) (see RFC 5007)
new-posix-timezone	41	string (RFC 4833)
new-tzdb-timezone	42	string (RFC 4833)
nis-domain-name*	29	DNS name
nis-servers*	27	IP address

Name (*=multivalue)	Number	Value
nisp-domain-name*	30	DNS name
nisp-servers*	28	IP address
oro*	6	unsigned 16-bit
pana agent	40	IPv6 address(es) (see RFC 5192)
preference	7	unsigned 8-bit
rapid-commit	14	zero size
reconfigure-accept	20	zero size
reconfigure-message	19	unsigned 8-bit
relay-agent-subscriber-id	38	binary
relay-message	9	binary
remote-id	37	binary; 2 fields: enterprise-id and remote-id
server-identifier	2	binary (AT_BLOB)
server-unicast	12	IPv6 address
sip-servers-address*	22	IPv6 address
sip-servers-name*	21	DNS name
sntp-servers*	31	IP address
status-code	13	binary; 2 fields: status-code and status-message
user-class*	15	counted-type; 2 fields: typecnt-size and user-data
vendor-class	16	vendor-class
vendor-opts	17	vendor-opts (see also cablelabs-17)

Request Dictionary

The table below lists the data items that you can set in the request dictionary at any time. The DHCP server reads them at various times. Unless indicated otherwise, all operations are read/write.

Table 5: Request Dictionary Specific Data Items

Data Item	Value (Protocol: v4=DHCPv4, v6=DHCPv6)
allow-bootp	int (v4)
If set to 1, allows BOOTP for any scope for this request. Read during scope selection and while checking for lease acceptability.	

Data Item	Value (Protocol: v4=DHCPv4, v6=DHCPv6)
allow-dhcp	int (v4)
If set to a 1, allows DHCP for any scope for this request. Read during scope selection and while checking for lease acceptability.	
allow-dynamic-bootp	int (v4)
If set to a 1, allows dynamic BOOTP for any scope for this request. Read during scope selection and while checking for lease acceptability.	
bootp-reply-options	blob (v4)
Overrides any *v4-bootp-reply-options* in any policy; read when gathering data for the output packet. (There are no IPv6 bootp-reply-options.)	
client-class-name	string (v4, v6)
Name of the client-class used to complete the client information (if any). Read-only.	
client-class-policy	string (v4, v6)
Name of the policy that is associated with the client-class. If set, it must be with the name of a policy that was already configured in the server.	
client-domain-name	string (v4, v6)
Domain name that the client wants to use. If it does not exist, in which case the DHCP server uses the domain name specified in the scope. Read when queuing the request for DNS update just prior to the update of stable storage. For DHCPv6, overrides the client-fqdn value and used for DNS updates.	
client-host-name	string (v4, v6)
Hostname for the client in DNS; read when queuing in the request for a DNS update just before updating stable storage. Places the actual name in DNS when that operation finishes. For DHCPv6, overrides the client-fqdn value and used for DNS updates.	
client-id	blob (v4, v6)
Client identification that the server uses to track the client. Can be the client-id sent with a request or internally generated from the MAC address. See *client-id-created-from-mac-address* . For DHCPv6, usually the client DUID.	
client-id-created-from-mac-address	int (v4)
If set to 1, the *client-id* must be created for internal use from the client-supplied MAC address and should not be used in reporting.	
client-ipaddress	IP address (v4)
IP address from which the client sent its packet. Note that it could be zero if the client does not yet have an IP address.	
client-limitation-id	blob (v4, v6)

Data Item	Value (Protocol: v4=DHCPv4, v6=DHCPv6)
Limitation ID for the client.	
client-lookup-id	blob (v4, v6)
Client lookup ID calculated by the *client-lookup-id* expression of the client-class.	
client-mac-address	blob (v4)
MAC address stored in the client object associated with the request dictionary. Has the same format (and was created from) the *mac-address* .	
client-os-type	int (v4)
Change the client entry of the request packet by setting this at the **pre-client-lookup** or **post-client-lookup** extension points. Can also be read at **check-lease-acceptable**, but cannot be set there. To set the value, you must first set the *os-type* in the **post-packet-decode** request dictionary.	
client-packet	blob (v4, v6, read-only)
The client portion of the received packet. For DHCPv4, this is the complete packet. For DHCPv6, this is the client message. (See packet to obtain the full packet.)	
client-policy	string (v4, v6)
Name of the policy that is associated with the client entry. If set, must be the name of a preconfigured policy in the DHCP server.	
client-port	int (v4, v6)
Port from which the client sent its request.	
client-requested-host-name	string (v4)
Hostname that the client requested be used for the DNS update. The DHCP server saves this information so that a change can be detected.	
client-unicast	boolean (v6, read-only)
True if the received packet was unicast by the client to the server.	
client-wants-nulls-in-strings	int (v4)
Determines whether the DHCP server returns strings to the client terminated with a null. If set to 1, the server terminates strings with a null. If set to 0, it does not terminate strings with a null. Set before **post-packet-decode** and read when encoding the response packet after **pre-packet-encode**.	
derived-vpn-id	int (v4, v6, read-only)
VPN identifier. See *vpn-name* for details.	
destination-ipaddress	IP address (v6, read-only)
Destination IPv6 address of the packet.	

Data Item	Value (Protocol: v4=DHCPv4, v6=DHCPv6)
dhcp-reply-options	blob (v4, v6)
Overrides any *v4-reply-options* or *v6-reply-options* specified in a policy; read when gathering data for the output packet.	
dump-packet	int (v4, v6, write-only)
When set to 1, Cisco Prime IP Express dumps the current decoded DHCP/BOOTP packet to the log file. An extension can put the value 1 into this data item at multiple points in its execution. This might be useful when debugging extensions.	
failover-role	int (v4, v6, read only)
Determines the failover server role. The failover server role can be one of three values: • • None—Failover is not configured. • Main/Backup—Failover is configured and the role of the failover server	
failover-state	int (v4, v6, read only)
Determines failover server state. The failover state can be normal, partner-down, communications-interrupted, recover, potential-conflict, recover-done, startup, shutdown, or paused. If failover is not configured the value is none.	
import-packet	int (v4)
Determines whether the server treats the packet as if it came from an import client. If set to 1, the server treats the client as an import client and performs all DNS operations on it before sending an ACK. Read when checking the server import mode (right after **post-packet-decode**), getting ready for DNS processing, and when setting the reply address.	
limitation-count	int (v4)
Number of simultaneous users allowed with the same *limitation-id*	
limitation-id	blob (v4)
Calculated by the *limitation-id* expression (if any) for the client-class in which this request falls	
limitation-id-null	int (v4, v6)
Set to 1(TRUE) if the *limitation-id* is null, 0 (FALSE) if another value.	
log-client-criteria-processing	int (v4, v6)
If set to a 1, logs the criteria processing for the client for this request. Read when trying to acquire a new lease for a client that does not have one, and when checking for lease acceptability.	
log-client-detail	int (v4, v6)
If set to a 1, logs the client-class processing for this request. Read at the end of client-class processing, after **post-client-lookup**.	

Data Item	Value (Protocol: v4=DHCPv4, v6=DHCPv6)
log-dns-update-detail	int (v4, v6)
If set to a 1, logs DNS update details for this request.	
log-dropped-bootp-packets	int (v4)
If set to a 1, logs dropped BOOTP packets for this request.	
log-dropped-dhcp-packets	int (v4, v6)
If set to a 1, logs dropped DHCP packets for this request.	
log-dropped-waiting-packets	int (v4, v6)
If set to a 1, logs dropped waiting packets for this request.	
log-failover-detail	int (v4)
If set to a 1, logs a more detailed level of failover activity, such as all failover state changes.	
log-incoming-packet-detail	int (v4, v6)
If set to a 1, checks whether detailed incoming packet tracing occurred for this request, so that you do not need to put a separate trace on it. Read before packet decoding and the first extension point.	
log-incoming-packets	int (v4, v6)
If set to a 1, logs the incoming packets for this request. Read after **post-decode-packet**	
log-ldap-create-detail	int (v4)
If set to a 1, logs messages whenever the DHCP server initiates a lease state entry creation to, receives a response from, or retrieves a result or error message from an LDAP server.	
log-ldap-query-detail	int (v4, v6)
If set to a 1, logs messages whenever the DHCP server initiates a query to, receives a response from, or retrieves a query result or an error message from an LDAP server.	
log-ldap-update-detail	int (v4)
If set to a 1, logs messages whenever the DHCP server initiates an update lease state to, receives a response from, or a retrieves a result or error message from an LDAP server.	
log-leasequery	int (v4, v6)
If set to a 1, logs messages when leasequery packets are processed without internal errors and result in an ACK or a NAK.	
log-missing-options	int (v4, v6)
If set to a 1, logs missing options (those a client requests but the DHCP server cannot return). Read while gathering data for the response.	

Data Item	Value (Protocol: v4=DHCPv4, v6=DHCPv6)
log-outgoing-packet-detail	int (v4, v6)
If set to a 1, logs a detailed dump of the outgoing packet for this request. Read after **pre-packet-encode** and just before sending the packet to the DHCP client.	
log-success-messages	int (v4, v6)
If set to a 1, logs the success messages.	
log-unknown-criteria	int (v4, v6)
If set to a 1, logs any unknown criteria specified in the client inclusion or exclusion criteria for this request. Read when acquiring a new client lease or checking lease acceptability for an existing client.	
log-v6-lease-detail	int (v6)
If set to 1, logs individual messages about DHCPv6 leasing activity.	
mac-address	blob (v4)
MAC address that came in the client packet. The first byte is the hardware type, the second is the hardware length, and the remaining (up to 16) is the information from the *chaddr* read just after **post-packet-decode**. This is a useful aggregation of the *htype* , *hlen* , and *chaddr* fields of the DHCP packet. When read it is constructed from these fields; when written it is placed into these fields.	
max-client-lookups	integer (v4, v6)
Maximum number of client database lookups allowed. Usually a small integer such as 2; the preset value is 1.	
override-client-id	blob (v4, v6)
Blob used for the current client-id value. Replaces any client-id from the incoming packet (although both values are kept in the lease state database).	
override-client-id-data-type	string (v4, v6, read-only)
Returns the data type of the *override-client-id* , either “nstr” for string or “blob” for blob.	
override-client-id-string	string (v4, v6)
Current client-id value in string format that replaces any client-id from the incoming packet (although both values are kept in the lease state database). For a *get* , if the *override-client-id* is not a string, the binary data is formatted as blob data, which is then returned as the “string.”	
packet	blob (v4, v6)
The received packet. For DHCPv4, this is the same as *client-packet* . For DHCPv6, this is the full packet if relayed or the same as *client-packet* if not relayed. It should only be written from the pre-packet-decode extension point; the server then decodes this new packet instead of the packet received from the client.	
ping-clients	int (v4)

Data Item	Value (Protocol: v4=DHCPv4, v6=DHCPv6)
If set to a 1, performs a ping before offering a lease for this request. Read just before determining if a lease is acceptable for a client.	
relay-agent-circuit-id	blob (v4, v6)
Contents of the circuit-id suboption of option 82.	
relay-agent-circuit-id-data	blob (v4, v6)
Contents of just the data part of the circuit-id suboption of option 82.	
relay-agent-device-class-data	blob (v4, v6)
Contents of the device-class suboption of option 82.	
relay-agent-radius-attributes	blob (v4)
Contents of the radius suboption of option 82.	
relay-agent-radius-class	string (v4)
Encapsulated class attribute of the radius suboption of option 82.	
relay-agent-radius-pool-name	string (v4)
Encapsulated framed-pool attribute of the radius suboption of option 82.	
relay-agent-radius-user	string (v4)
Encapsulated user attribute of the radius suboption of option 82.	
relay-agent-remote-id	blob (v4, v6)
Contents of the remote-id suboption of option 82.	
relay-agent-remote-id	blob (v4, v6)
Contents of just the data part of the remote-id suboption of option 82.	
relay-agent-server-id-override-data	IPv6 address (v4, v6)
Contents of the server-id suboption of option 82. If the IANA suboption 182 is in the packet, that value appears; otherwise, the Cisco suboption 152 value appears.	
relay-agent-subscriber-id	string (v4)
Contents of the subscriber-id suboption of option 82	
relay-count	int (v6, read-only)
Number of DHCPv6 relay hops	
reply-options	blob

Data Item	Value (Protocol: v4=DHCPv4, v6=DHCPv6)
Overrides any DHCPv4 reply options specified in any policy. Read when gathering data for the output packet.	
reply-to-client-address	int (v4, v6)
For v4, if set to 1, the server sends the response packet to the client-ipaddress and the client-port. For v6, if set to 1, the server sends the response packet back to the address and port of the sender (client or relay agent). If 0, the server sends the response using the RFC mandated algorithm.	
reserved-addresses	IP address (v4, read/write)
List of addresses reserved for the client. The first available address to match a usable Scope (which must have restrict-to-reservations enabled) will be assigned to the client.	
reserved-ip6addresses	IP address (v6, read/write)
List of addresses reserved for the client. All available addresses to match a usable Prefix (which must have restrict-to-reservations enabled) will be assigned to the client.	
reserved-prefixes	IP address (v6, read/write)
List of prefixes reserved for the client. All available prefixes to match a usable Prefix (which must have restrict-to-reservations enabled) will be assigned to the client.	
selection-criteria	string (v4, v6)
Comma-separated string that contains the scope selection criteria.	
selection-criteria-excluded	string (v4, v6)
Comma-separated string that contains the scope exclusion criteria.	
send-ack-first	int (v4, v6)
If set to a 1, updates DNS after the ACK for DHCP requests. Read just before initiating the DNS operation.	
source-ipaddress	IPv6 address (v6, read-only)
IPv6 source address of the packet.	
trace-id	string (v4, v6, read-only)
ID used by the system to trace the packet.	
transaction-time	int (v4, v6)
Time, in seconds since 1970, that the input packet was decoded.	
update-dns	string (v4, v6)
Requests partial, full, or no dynamic DNS updates on a per-request packet basis. Input and output values are: 1=update-all, 2=update-fwd-only, 3=update-rev-only, and 0=update-none.	
update-dns-for-bootp	int (v4)

Data Item	Value (Protocol: v4=DHCPv4, v6=DHCPv6)
If set to a 1, updates DNS for BOOTP requests for this request. Read just before initializing the DNS operation for BOOTP.	
verbose-logging	int (v4, v6)
If set to a 1, logs verbose messages for this request. Read at various times during processing.	
vpn-description	string (v4, v6, read-only)
Description for the VPN. See *vpn-name* for details.	
vpn-name	string (v4, v6, read-only)
Name of the VPN. The request dictionary does not have valid values for these items at **post-packet-decode**, but does at all other extension points, because the VPN has not yet been determined. This is so that a script can change the *derived-vpn-id* option or suboption at **post-packet-decode** and thereby affect the VPN used for a lease	
vpn-vpn-id	blob, typically 7 bytes (v4, v6, read-only)
Virtual private network identifier. See *vpn-name* for details.	
vpn-vrf-name	string (v4, v6, read-only)
Virtual routing and forwarding table identifier for the VPN. See *vpn-name* for details.	

Response Dictionary

The table below lists the data items you can set in the response dictionary at any time. The DHCP server reads them at various times. Unless indicated otherwise, the operation is read/write.

Table 6: Response Dictionary Specific Data Items

Data Item	Value (Protocol: v4=DHCPv4, v6=DHCPv6)
client-active-lease-count	int (v6, read-only)
Number of active leases on the DHCPv6 client.	
client-creation-time	int (v6, read-only)
Creation time of the IPv6 client.	
client-domain-name	string (v4, read-only)
From the client information in the lease, the domain name that the client wants to use. It might not exist, in which case the DHCP server uses the domain name specified in the scope. Read when queuing the request for DNS update just prior to the update of stable storage.	
client-expiration-time	int (v4, v6, read-only)

Data Item	Value (Protocol: v4=DHCPv4, v6=DHCPv6)
The highest lease expiration time given to the client by this server (in seconds, since 1970).	
client-host-name	string (v4, read-only)
From the client information in the lease, the hostname that the DHCP server puts into DNS. Read when queueing the request for a DNS update just before updating stable storage.	
client-id	blob (v4, v6, read-only)
From the client information in the lease, the client identification that the server used to keep track of the client. This might be the client-id sent with a request or internally generated from the MAC address.For DHCPv6, usually the client DUID.	
client-id-created-from-mac-address	int (v4, read-only)
From the client information in the lease. If set to 1, the *client-id* must be created from the MAC address and should not be used in reporting.	
client-last-transaction-time	int (v4, v6, read-only)
Time, in seconds, since 1970, that the DHCP server last heard from this client.	
client-limitation-id	blob (v4, read-only)
Limitation identifier of the client associated with the current lease.	
client-mac-address	blob (v4, read-only)
From the client information in the lease, the MAC address stored in the client object associated with the request dictionary. Has the same format as (and was created from) the *mac-address* .	
client-os-type	int (v4)
Change the client entry of the request packet by setting this at the **pre-client-lookup** or **post-client-lookup** extension points. Can also be read at **check-lease-acceptable**, but cannot be set there. To set the value, you must first set the *os-type* in the **post-packet-decode** request dictionary.	
client-override-client-id	blob (v4, v6, read-only)
Blob used for the current client-id value. Replaces any client-id from the incoming packet (although both values are kept in the lease state database).	
client-override-client-id-data-type	string (v4, v6, read-only)
Returns the data type of the *client-override-client-id* , either nstr for string or *blob* for blob.	

Data Item	Value (Protocol: v4=DHCPv4, v6=DHCPv6)
client-override-client-id-string	string (v4, v6, read-only)
Current client-id value in string format that replaces any client-id from the incoming packet (although both values are kept in the lease state database). For a *get* , if the *client-override-client-id* is not a string, the binary data is formatted as blob data, which is then returned as the “string.”	
client-packet	blob (v4, v6, read-only)
The client portion of the response packet. For DHCPv4, this is the complete packet. For DHCPv6, this is the client message. (See packet to obtain the full packet.) Only available from the post-packet-encode extension point.	
client-reconfigure-key	string (v6)
Returns the *client-reconfigure-key* attribute value of the DHCPv6 lease.	
client-reconfigure-key- generation-time	string (v6)
Returns the *client-reconfigure-key-generation-time* attribute value of the DHCPv6 lease.	
client-relay-address	IPv6 address (v6, read-only)
Source IPv6 address for the (last) relay.	
client-relay-message	string (v6, read-only)
Last relayed DHCPv6 message, excluding the client message.	
client-requested-host-name	string (v4)
From the client information in the lease, the hostname that the client requested for the DNS update.	
client-user-defined-data	string (v4, v6, read-only)
Returns the value previously or currently associated with the client, as derived from the *user-defined-data* environment dictionary data item. It returns the previously associated value if requested in a **check-lease-acceptable** or **lease-state-change** extension point. It returns the current value if requested in a **pre-packet-encode** or **post-send-packet** extension point.	
client-vendor-class	string (v4, v6)
Returns the *client-vendor-class* attribute value of the DHCPv4 or DHCPv6 lease.	
client-vendor-info	string (v4, v6)

Data Item	Value (Protocol: v4=DHCPv4, v6=DHCPv6)
Returns the *client-vendor-info* attribute value of the DHCPv4 or DHCPv6 lease.	
client-write-sequence	int (v6, read-only)
Write sequence of the client IPv6 request.	
client-write-time	int (v6, read-only)
Time of the client IPv6 write request.	
derived-vpn-id	int (v4, v6, read-only)
VPN identifier.	
domain-name-changed	int (v4)
If set to 1, the domain name in the current packet differs from the domain name used in the DNS update. Read after **check-lease-acceptable** and before **pre-packet-encode**.	
dump-packet	int (v4, v6, write-only)
When set to 1, Cisco Prime IP Express dumps the current decoded DHCP/BOOTP packet to the log file. An extension can put the value 1 into this data item at multiple points in its execution. This might be useful when debugging extensions.	
failover-role	int (v4, v6, read only)
Determines the failover server role. The failover server role can be one of three values: • • None—Failover is not configured. • Main/Backup—Failover is configured and the role of the failover server	
failover-state	int (v4, v6, read only)
Determines failover server state. The failover state can be normal, partner-down, communications-interrupted, recover, potential-conflict, recover-done, startup, shutdown, or paused. If failover is not configured the value is none.	
host-name-changed	int (v4)
If set to 1, the hostname in the current packet differs from that used in the DNS update. Read after **check-lease-acceptable** and before **pre-packet-encode**.	
host-name-in-dns	int (v4, v6)

Data Item	Value (Protocol: v4=DHCPv4, v6=DHCPv6)
If set to 1, the hostname is in DNS. Read after **check-lease-acceptable** and before **pre-packet-encode**. Written after the hostname goes into DNS.	
lease-binding-iaid	int (v6, read-only)
IPv6 lease binding IAID.	
lease-binding-rebinding-time	int (v6, read-only)
IPv6 lease binding rebinding time.	
lease-binding-renewal-time	int (v6, read-only)
IPv6 lease binding renewal time.	
lease-binding-type	string (v6, read-only)
IPv6 lease binding type: "IA_NA", "IA_TA", or "IA_PD".	
lease-client-reserved	int (v4, v6, read-only)
Returns 1 if the lease is client reserved and 0 if not.	
lease-creation-time	string (v6, read-only)
IPv6 lease creation time.	
lease-deactivated	int (v4, v6, read-only)
If set to 1, reports that the lease is deactivated.	
lease-dns-forward-backup-server-address	IP address (v4, v6, read-only)
Address of the backup DNS server that receives DNS updates for the DHCPv4 and DHCPv6 lease, if the server specified in *lease-dns-forward-server-address* is down.	
lease-dns-forward-server-address	IP address (v4, v6, read-only)
Address of the DNS server that receives dynamic DNS updates for the DHCPv4 and DHCPv6 lease.	
lease-dns-forward-update	string (v4, v6, read-only)
Name of the update configuration that determines the forward zones to be included in DNS updates for the DHCPv4 and DHCPv6 lease. Returns TRUE if *update-all* or *update-fwd-only* is set.	
lease-dns-forward-zone-name	string (v4, v6, read-only)
Name of an optional forward zone for DNS updates.	
lease-dns-reverse-backup-server-address	IP address (v4, v6, read-only)

Data Item	Value (Protocol: v4=DHCPv4, v6=DHCPv6)
Address of the backup DNS server that receives DNS updates for a DHCPv4 and DHCPv6 lease, if the server specified in *lease-dns-reverse-server-address* is down.	
lease-dns-reverse-host-bytes	int (v4, read-only)
The number of bytes in a lease IP address to use for a reverse zone.	
lease-dns-reverse-prefix-length	int (v6, read-only)
Prefix length of the reverse zone for ip6.arpa updates.	
lease-dns-reverse-server-address	IP address (v4, v6, read-only)
Address of the DNS server address that receives dynamic DNS updates for the DHCPv4 and DHCPv6 lease.	
lease-dns-reverse-update	string (v4, v6, read-only)
Name of the update configuration that determines which reverse zones to include in a DNS update for the DHCPv4 and DHCPv6 lease. Returns TRUE if *update-all* or *update-fwd-only* is set.	
lease-dns-reverse-zone-name	string (v4, v6, read-only)
DNS reverse (in-addr.arpa and ip6.arpa) zone that is updated with PTR records.	
lease-fqdn	string (v6, read-only)
Fully qualified domain name assigned to the DHCPv6 lease by the server (and possibly successfully entered into DNS). The *lease-fqdn* may be the name that is expected to be added to DNS for the lease or the actual name added. If *host-name-in-dns* is equal to true, the actual *lease-fqdn* is in DNS.	
lease-requested-fqdn	string (v6, read-only)
Partial or fully qualified domain name most recently requested by the client for the DHCPv6 lease.	
lease-giaddr	IP address (v4, read-only)
Lease *giaddr* .	
lease-ipaddress	IPv4 or IPv6 address or prefix (v4, v6, read-only)
For DHCPv4, the address of the lease associated with the client. For DHCPv6, the IPv6 address or IPv6 prefix (address and prefix-length) of the lease for the current context (See setObject method).	

Data Item	Value (Protocol: v4=DHCPv4, v6=DHCPv6)
lease-preferred-lifetime	int (v6, read-only)
Preferred lifetime of the IPv6 lease.	
lease-prefix-name	string (v6, read-only)
Prefix name of the IPv6 lease.	
lease-relay-agent-info	blob (v4)
Entire contents of option 82.	
lease-relay-agent-circuit-id	blob (v4)
Accesses and manipulates the relay agent circuit ID as stored with the lease of a response. Requires the suboption number 1 as the first byte. Deprecated in favor of the *lease-relay-agent-circuit-id-data* item.	
lease-relay-agent-circuit-id-data	blob (v4, use instead of deprecated *lease-relay-agent-circuit-id*)
Accesses and manipulates the *relay-agent-circuit-id-data* as stored with the lease of a response.	
lease-relay-agent-device-class-data	blob (v4)
Contents of the device-class suboption of option 82.	
lease-relay-agent-radius-attributes	blob (v4)
Contents of the radius suboption of option 82.	
lease-relay-agent-radius-class	string (v4)
Encapsulated class attribute of the radius suboption of option 82.	
lease-relay-agent-radius-pool-name	string (v4)
Encapsulated framed-pool attribute of the radius suboption of option 82.	
lease-relay-agent-radius-user	string (v4)
Encapsulated user attribute of the radius suboption of option 82.	
lease-relay-agent-remote-id	blob (v4)
Accesses and manipulates the *relay-agent-remote-id* data as stored with the lease of a response. Requires suboption number 2 as the first byte. Deprecated in favor of the *lease-relay-agent-remote-id-data* item.	

Data Item	Value (Protocol: v4=DHCPv4, v6=DHCPv6)
lease-relay-agent-remote-id-data	blob (v4, use instead of *lease-relay-agent-remote-id* item)
Accesses and manipulates the *relay-agent-remote-id-data* as stored with the lease of a response.	
lease-relay-agent-server-id- override-data	IP address (v4)
Accesses and manipulates the *relay-agent-server-id-override-data* as stored with the lease of a response.	
lease-relay-agent-subnet- selection-data	IP address (v4)
Accesses and manipulates the *relay-agent-subnet-selection-data* as stored with the lease of a response.	
lease-relay-agent-subscriber-id	string (v4)
Contents of the subscriber-id suboption of option 82.	
lease-relay-agent-vpn-id-data	blob (v4)
Accesses and manipulates the *relay-agent-vpn-id* data as stored with the lease of a response.	
lease-reserved	int (v4, v6, read-only)
Returns 1 if the lease is lease reserved and 0 if not.	
lease-start-time-of-state	int (v4, v6, read-only)
Time, in seconds since 1970, that this lease was first placed into its current state.	
lease-state	string (v4, v6, read-only)
State of the lease, which can be available, offered, leased, expired, unavailable, released, other-available (DHCPv4 only), pending-available (DHCPv4 only), or revoked (DHCPv6 only).	
lease-state-expiration-time	int (v6, read-only)
Expiration time of the IPv6 lease state.	
lease-status	string (v4, v6, read-only)
Returns “nonexistent,” “owned-by-client,” or “exists.” Used to determine if a lease exists and if the current client owns it. If “exists” is returned, the lease exists but the current owner does not own it (limited information on the lease is available).	

Data Item	Value (Protocol: v4=DHCPv4, v6=DHCPv6)
lease-valid-lifetime	int (v6, read-only)
Valid lifetime of the IPv6 lease.	
lease-vpn-description	string (v4, v6, read-only)
Description for the VPN stored with the lease of a response.	
lease-vpn-id	int (v4, v6, read-only)
Identifier for the VPN stored with the lease of a response.	
lease-vpn-name	string (v4, v6, read-only)
Name of the VPN stored with the lease of a response.	
lease-vpn-vpn-id	blob, typically 7 bytes (v4, v6, read-only)
Virtual private network (VPN) identifier stored with the lease of a response.	
lease-vpn-vrf-name	string (v4, v6, read-only)
Virtual routing and forwarding table identifier for the VPN stored with the lease of a response.	
mac-address	blob (v4)
MAC address that came in the client packet. The first byte is the hardware type, the second is the hardware length, and the remaining (up to 16) is the information from the *chaddr* . This is a useful aggregation of the *htype* , *hlen* , and *chaddr* fields of the DHCP packet. When read it is constructed from these fields; when written it is placed into these fields.	
override-client-id	blob (v4, v6, read-only)
Blob used for the current client-id value. Replaces any client-id from the incoming packet (although both values are kept in the lease state database).	
override-client-id-data-type	string (v4, v6, read-only)
Returns the data type of the *override-client-id* , either “nstr” for string or “blob” for blob.	
override-client-id-string	string (v4, v6, read-only)
Current client-ID value in string format that replaces any client-id from the incoming packet (although both values are kept in the lease state database). For a *get* , if the *override-client-id* is not a string, the binary data is formatted as blob data, which is then returned as the “string.”	

Data Item	Value (Protocol: v4=DHCPv4, v6=DHCPv6)
packet	blob (v4, v6, use only at **post-packet-decode**)
The response packet. For DHCPv4, this is the same as client-packet. For DHCPv6, this is the full packet if relayed or the same as client-packet if not relayed. It should only be read or written from the post-packet-encode extension point; if written, the server will then send the new packet to the client.	
ping-clients	int (v4)
If set to 1, performs a ping before offering a lease for this request. Read just before determining a client lease acceptability.	
prefix-address	IPv6 prefix (v6, read-only)
Prefix address (17 bytes—IPv6 address and prefix length).	
prefix-allocate-random	int (v6, read-only)
Prefix randomly allocated.	
prefix-allocate-via-best-fit	int (v6, read-only)
Prefix allocated via the best fit.	
prefix-allocate-via-client-request	int (v6, read-only)
Prefix allocated via client request.	
prefix-allocate-via-extension	int (v6, read-only)
Prefix allocated via an extension.	
prefix-allocate-via-reservation	int (v6, read-only)
Prefix allocated via a reservation.	
prefix-allocate-via-interface- identifier	int (v6, read-only)
Prefix allocated via an interface identifier.	
prefix-allocation-group	string (v6, read-only)
Allocation group name for the prefix.	
prefix-allocation-group-priority	int (v6, read-only)
Allocation group priority for the prefix.	
prefix-deactivated	int (v6, read-only)
Indicates if the prefix is deactivated.	

Data Item	Value (Protocol: v4=DHCPv4, v6=DHCPv6)
prefix-dhcp-type	string (v6, read-only)
Prefix DHCP type.	
prefix-expiration-time	string (v6, read-only)
Expiration time of the prefix.	
prefix-link-group-name	string (v6, read-only)
Link group name for the link.	
prefix-link-name	string (v6, read-only)
Link of the prefix.	
prefix-link-type	string (v6, read-only)
Link type (topological, location-independent, or universal).	
prefix-name	string (v6, read-only)
Name of the prefix.	
prefix-range	IPv6 address (v6, read-only)
IPv6 address range of the prefix.	
prefix-restrict-to-reservations	int (v6, read-only)
If set to 1, the prefix has restrict-to-reservations enabled.	
prefix-selection-tags	string (v6, read-only)
Selection tags of the prefix.	
relay-count	int (v6, read-only)
Number of DHCPv6 relay hops.	
reply-ipaddress	IPv4 or IPv6 address (v4, v6)
IP address to use when replying to the DHCP client. Read just after **pre-packet-encode**. If you change its value in a **pre-packet-encode**, the IP address you place in it should be for a system that can respond to ARP queries (unless it is a broadcast address). Even if unicast is enabled and the broadcast flag is not set in the DHCP request, the local ARP cache is not set with a mapping from a new *reply-ipaddress* in the **pre-packet-encode** to the MAC address in the DHCP request.	
reply-port	int (v4, v6)

Data Item	Value (Protocol: v4=DHCPv4, v6=DHCPv6)
Port to use when replying to the DHCP client. Read just after **pre-packet encode**.	
response-source	string (v4, v6, read-only)
The source of the response (the major activity that invoked the extension). Output values are: client (Received client packet), failover (Received binding update from the failover partner), timeout (Lease expiration or grace period end), operator (Request from a user interface), one-lease-per-client (One lease per client removing a client from an old lease because of a new one), unknown (None of the above). This data item helps an extension to determine what processing it should do whether a request dictionary is present or not. (The isValid method can also be used to determine whether a dictionary is valid.)	
reverse-name-in-dns	int (v4, v6)
If equal to 1, the reverse name is in DNS. Read before initializing a DNS operation.	
scope-allow-bootp	int (v4, read-only)
If set to 1, the scope allows BOOTP. Written after a DNS operation finishes.	
scope-allow-dhcp	int (v4, read-only)
If set to 1, the scope allows DHCP.	
scope-allow-dynamic-bootp	int (v4, read-only)
If set to 1, the scope allows dynamic BOOTP.	
scope-available-leases	int (v4, read-only)
Number of available leases on the current scope.	
scope-deactivated	int (v4, read-only)
If set to 1, the scope is deactivated.	
scope-dns-forward-server-address	IP address (v4, read-only)
DNS server to use for the DNS forward address.	
scope-dns-forward-zone-name	string (v4, read-only)
Forward zone name configured in the scope.	
scope-dns-number-of-host-bytes	int (v4, read-only)
Number of host bytes used by the DHCP server code that handles DNS updates.	

Data Item	Value (Protocol: v4=DHCPv4, v6=DHCPv6)
scope-dns-reverse-server-address	IP address (v4, read-only)
DNS server to use for the DNS reverse address.	
scope-dns-reverse-zone-name	string (v4, read-only)
Reverse zone name configured in the scope.	
scope-network-number	IP address (v4, read-only)
Network number of the scope that contains the lease the DHCP server is processing.	
scope-ping-clients	int (v4, read-only)
If set to 1, the scope associated with the current lease was configured to support a ping operation prior to offering a lease.	
scope-primary-network-number	IP address (v4, read-only)
Network number of this primary scope.	
scope-primary-subnet-mask	IP address (v4, read-only)
Subnet mask of this primary scope.	
scope-renew-only	int (v4, read-only)
If set to 1, the scope is renew-only.	
scope-renew-only-expire-time	int (v4, read-only)
Absolute time, in seconds since January 1, 1970, at which a renew-only scope should cease to be renew-only.	
scope-restrict-to-reservations	int (v4, read-only)
If set to 1, the scope has restrict-to-reservations enabled.	
scope-selection-tags	string (v4, read-only)
Comma-separated string that contains the scope selection criteria. Use this data item for decisions based on scopes.	
scope-send-ack-first	int (v4, read-only)
If set to 1, the scope sends an ACK before performing the rest of the processing.	
scope-subnet-mask	IP address (v4, read-only)
Subnet mask of the scope that contains the lease the DHCP server is processing.	

Data Item	Value (Protocol: v4=DHCPv4, v6=DHCPv6)
scope-update-dns	string (v4, read-only)
DNS updates for forward or reverse zones. Output values are: 1=update-all, 2=update-fwd-only, 3=update-rev-only, and 0=update-none.	
scope-update-dns-enabled	boolean (v4, read-only)
If set to 1, the scope has update DNS enabled for forward and reverse zones. Deprecated in favor of *scope-update-dns*.	
scope-update-dns-for-bootp	int (v4, read-only)
If set to 1, the scope has update DNS enabled for BOOTP.	
trace-id	string (v4, v6, read-only)
ID used by the system to trace the packet.	
transaction-time	int (v4, v6, read-only)
Time, in seconds since 1970, that the request was decoded.	
vpn-description	string (v4, v6, read-only)
Description for the VPN.	
vpn-name	string (v4, v6, read-only)
Name of the VPN.	
vpn-vpn-id	blob, typically 7 bytes (v4, v6, read-only)
Virtual private network (VPN) identifier.	
vpn-vrf-name	string (v4, v6, read-only)
Virtual routing and forwarding table (VRF) identifier for the VPN.	

Extension Dictionary API

This section contains the dictionary method calls to use when accessing dictionaries from Tcl extensions and shared libraries.

Tcl Attribute Dictionary API

In an attribute dictionary, the keys are constrained to be the names of attributes as defined in the Cisco IP Express DHCP server configuration. The values are the string representation of the legal values for that particular attribute. For example, IP addresses are specified by the dotted-decimal string representation of the

address, and enumerated values are specified by the name of the enumeration. This means that numbers are specified by the string representation of the number.

Attribute dictionaries are unusual in that they can contain more than one instance of a key. These instances are ordered, with the first instance at index zero. Some of the attribute dictionary methods allow an index to indicate a particular instance or position in the list of instances to be referenced.

Tcl Request and Response Dictionary Methods

Attribute dictionaries use commands with which you can change and access the values in the dictionaries. The table below lists the commands to use with the request and response dictionaries. In this case, you can define the *dict* variable as **request** or **response**.

See the *install-path* /examples/dhcp/tcl/tclextension.tcl file for examples.

Table 7: Tcl Request and Response Dictionary Methods

Method	Syntax
get Returns the value of the attribute from the dictionary, represented as a string. If the dictionary does not contain the attribute, the empty string is returned instead. If you include the index value, this returns the *index* th instance of the attribute. Some attributes can appear more than once in the request or response packet. The index selects which instance to return. If you include the *bMore* , the **get** method sets *bMore* to TRUE if there are more attributes after the one returned, otherwise to FALSE. Use this to determine whether to make another call to **get** to retrieve other instances of the attribute.	**$***dict* **get** *attribute* [*index* [*bMore*]]
getOption Gets the data for an option as a string. See Table 8: Tcl arg-type and obj-type Values , on page 36 for the *arg-type* values. If the next argument is a numeric value, it is assumed to be a number, otherwise a name. Note that this function always returns a pointer to a string, which can be zero length if the option does not exist or has length zero. For sample usage, see the Handling Vendor Class Option Data, on page 52.	**$***dict* **getOption** *arg-type* [*arg-data*]
isValid isV4 isV6	**$***dict* **isValid $***dict* **isV4 $***dict* **isV6**

Method	Syntax
The **isValid** method returns TRUE if there is a request or response (depending on the dictionary passed in); FALSE otherwise. Extensions such as **lease-state-change** can use this method to determine whether a dictionary is available. The **isV4** method returns TRUE if this extension is being called for a DHCPv4 packet; FALSE otherwise. Calling this method from an **init-entry** routine returns FALSE. The **isV6** method returns TRUE if this extension is being called for a DHCPv6 packet; FALSE otherwise. Calling this method from an **init-entry** routine returns FALSE.	
log	**$***dict* **log** *level message ...*
Puts a message into the DHCP server logging system. The level should be LOG_ERROR, LOG_WARNING, or LOG_INFO. The remaining arguments are concatenated and sent to the logging system at the specified level. **Note** Use the LOG_ERROR and LOG_WARNING levels sparingly, because the server flushes its log file with messages logged at these levels. Using these levels for messages that are likely to occur frequently (such as client requests) can have severe impact on disk I/O performance.	
moveToOption	**$***dict* **moveToOption** *arg-type* [*arg-data*] ...
Sets the context for subsequent **get**, **put**, and **remove** option operations. See Table 8: Tcl arg-type and obj-type Values , on page 36 for the *arg-type* values. Note that the context can become invalid if the option is removed (such as by **removeOption**).	
put	**$***dict* **put** *attribute value* [*index*]
Associates a value with the attribute in the dictionary. If you omit the index or set it to the special value REPLACE, this replaces any existing instances of the attribute with the single value. If you include the index value as the special value APPEND, this appends a new instance of the attribute to the end of the list of instances of the attribute. If you include the index value as a number, this inserts a new instance of the attribute at the position indicated. If you set the index value to the special value AUGMENT, this puts the attribute only if there is not one already.	
putOption	**$***dict* **putOption** *data arg-type* [*arg-data*] ...
Adds an option and its data or modifies the data for an option. See Table 8: Tcl arg-type and obj-type Values , on page 36 for the *arg-type* values. For sample usage, see the Handling Vendor Class Option Data, on page 52.	
remove	**$***dict* **remove** *attribute* [*index*]

Method	Syntax
Removes the attribute from the dictionary. If you omit the index or set it to the special value REMOVE_ALL, this removes any existing instances of the attribute. If you include the index as a number, this removes the instance of the attribute at the position indicated. This method always returns 1, even if the dictionary does not contain that attribute at that index.	
removeOption	**$***dict* **removeOption** *arg-type* [*arg-data*] ...
Removes an option. See Table 8: Tcl arg-type and obj-type Values , on page 36 for the *arg-type* values. For sample usage, see the Handling Vendor Class Option Data, on page 52.	
setObject	**$***dict* **setObject** *obj-type* [*data*]
(DHCPv6 only.) Sets the object for **get**, **put**, and **remove** methods, and alters the message on which the new option methods operate. See Table 8: Tcl arg-type and obj-type Values , on page 36 for the *obj-type* values. DHCPv6 extensions primarily use this method to access the leases and prefixes available for the client and link, or to get message header fields or options from relay packets. Unlike in DHCPv4, where one lease and scope are associated with a response, a DHCPv6 response can involve several leases and prefixes. Returns TRUE if the object exists; FALSE otherwise. For sample usage, see the Handling Object Data, on page 53. **Note** For leases not associated with the current client, only minimal information is available.	
trace	**$***dict* **trace** *level message ...*
Returns a message in the DHCP server packet tracing system. At level 0, no tracing occurs. At level 1, it traces only that the server received the packet and sent a reply. At level 4, it traces everything. The remaining arguments are concatenated and sent to the tracing system at the specified level. The default tracing is set using the DHCP server *extension-trace-level* attribute.	

Table 8: Tcl arg-type and obj-type Values

arg-type or obj-type	Description
enterprise *number/name*	Enterprise-id number or name for the option definition set for the option or suboption.
home	Requests that the context is reset to the “top” of the current client or relay message.
index *number/keyword*	Number or keyword (replace, append, augment, raw, or remove_all) for the array index on which to operate.
index-count	Returns the number of array index entries in the option.

arg-type or obj-type	Description
instance *number*	Instance number of the option (primarily used for DHCPv6).
instance-count	Returns the number of times the option appears (if 0, the option is not present).
more *tcl-variable-name*	Name of a Tcl variable that is set to TRUE or FALSE, depending on whether more array index entries exist in the option data.
move-to	Requests that the context be set to the option.
option *number/name*	Option number or name to operate on.
parent	Requests that the context is moved up one option.
suboption *number/name*	Suboption number or name to operate on.
vendor *name*	Vendor name for the option definition set for the option or suboption.
lease initial | *index* | *address* | *prefix*	Used with **setObject**, sets the context for the lease, binding, and prefix data items in the response dictionary to the indicated lease. The **initial** keyword requests that the original context for when the extension was called is restored. The *index* requests that the numbered lease (starting at 0) is set and can be used to iterate through all of the leases for the client. The *address* or *prefix* requests that the lease with that address or prefix is set (if it exists).
message initial | *number*	Used with **setObject**, sets the context for the message data items and options in the request or response dictionary to the indicated message. The **initial** keyword sets the context to the client message. The *number* sets the context to the relay message, with 0 specifying the relay closest to the client.
prefix initial | *index* | *address* | *prefix* | *name*	Used with **setObject**, sets the context for the prefix data items in the response dictionary to the indicated prefix. The **initial** keyword requests that the original context for when the extension was called is restored. The *index* requests the numbered prefix (starting at 0) is set and can be used to iterate through all of the prefixes for the client on the link. The *address* or *prefix* requests that the prefix for the address or prefix is set (if found). The *name* requests that the named prefix is found. Note that only prefixes on the current link can be used.

Tcl Environment Dictionary Methods

The table below describes the commands to use with the environment dictionary. In this case, you can define the *dict* variable as **environ**, as in the following procedure example:

```
proc tclhelloworld2 { request response environ } {
$environ put trace-level 4
$environ log LOG_INFO "Environment hello world"
}
```

Table 9: Tcl Environment Dictionary Methods

Method	Syntax
clear	**$***dict* **clear**
Removes all entries from the dictionary.	
containsKey	**$***dict* **containsKey** *key*
Returns 1 if the dictionary contains the key, otherwise 0.	
firstKey	**$***dict* **firstKey**
Returns the name of the first key in the dictionary. Note that the keys are not stored sorted by name. If a key does not exist, returns the empty string.	
get	**$***dict* **get** *key*
Returns the value of the key from the dictionary. If a key does not exist, returns the empty string.	
isEmpty	**$***dict* **isEmpty**
Returns 1 if the dictionary has no entries, otherwise 0.	
log	**$***dict* **log** *level message ...*
Returns a message in the DHCP server logging system. The *level* should be one of LOG_ERROR, LOG_WARNING, or LOG_INFO. The remaining arguments are concatenated and sent to the logging system at the specified level. **Note** Use the LOG_ERROR and LOG_WARNING levels sparingly, because the server flushes its log file with messages logged at these levels. Using these levels for messages that are likely to occur frequently (such as client requests) can have severe impact on disk I/O performance.	
nextKey	**$***dict* **nextKey**
Returns the name of the next key in the dictionary that follows the key returned in the last call to **firstKey** or **nextKey**. If a key does not exist, returns the empty string.	
put	**$***dict* **put** *key value*
Associates a value with the key, replacing an existing instance of the key with the new value.	
remove	**$***dict* **remove** *key*
Removes the key from the dictionary. Always returns 1, even if the dictionary did not contain the key.	
size	**$***dict* **size**
Returns the number of entries in the dictionary.	
trace	**$***dict* **trace** *level message ...*

Method	Syntax
Returns a message in the DHCP server packet tracing system. At level 0, no tracing occurs. At level 1, it traces only that the server received the packet and sent a reply. At level 4, it traces everything. The remaining arguments are concatenated and sent to the tracing system at the specified level. The default tracing is set using the DHCP server *extension-trace-level* attribute.	

DEX Attribute Dictionary API

When writing DEX extensions for C/C++, you can specify keys as the attribute name string representation or by type (a byte sequence defining the attribute). This mean that some of these access methods have four different variations that are the combinations of string or type for the key or value.

A basic DEX extension example might be:

```
int DEXAPI dexhelloworld( int iExtensionPoint,
dex_AttributeDictionary_t *pRequest,
dex_AttributeDictionary_t *pResponse,
dex_EnvironmentDictionary_t *pEnviron )
{
pEnviron->log( pEnviron, DEX_LOG_INFO, "hello world" );
return DEX_OK;
}
```

See the *install-path* /examples/dhcp/dex/dexextension.c file or other files in that directory for examples.

DEX Request and Response Dictionary Methods

DEX attribute dictionaries use active commands, called methods, with which you can change and access values. The table below lists the methods to use with the request and response dictionaries. In this case, you can define the *pDict* variable as **pRequest** or **pResponse**, as in:

```
pRequest->get( pRequest, "host-name", 0, 0 );
```

The *pszAttribute* is the **const char *** pointer to the attribute name that the application wants to access. The *pszValue* is the pointer to the **const char *** string that represents the data (returned for a **get** method, and stored in a **put** method). See the corresponding *iObjectType* , *iObjArgType* , and *iArgType* tables, respectively.

Tip See also the Differences in get, put, Option, Bytes, and OptionBytes Methods, on page 45 and the Differences in get, put, remove, and ByType Methods, on page 45.

Table 10: DEX Request and Response Dictionary Methods

Method	Syntax
allocateMemory	**void *** *pDict* **->allocateMemory(dex_AttributeDictionary_t *** *pDict*, **unsigned int** *iSize* **)**
Allocates memory in extensions that persists only for the lifetime of this request.	

Method	Syntax
get	**const char ****pDict***->get(dex_AttributeDictionary_t ****pDict***, const char ****pszAttribute***, int** *iIndex***, abool_t ****pbMore* **)**
Returns the value of the *iIndex* ed instance of the attribute from the dictionary, represented as a string. If the dictionary does not contain the attribute (or that many instances of it), the empty string is returned instead. If *pbMore* is nonzero, the **get** method sets *pbMore* to TRUE if there are more instances of the attribute after the one returned, otherwise to FALSE. Use this to determine whether to make another call to **get** to retrieve other instances of the attribute.	
getBytes	**const abytes_t ****pDict***->getBytes(dex_AttributeDictionary_t ****pDict***, const char ****pszAttribute***, int** *iIndex***, abool_t ****pbMore* **)**
Returns the value of the *iIndex* ed instance of the attribute from the dictionary as a sequence of bytes. If the dictionary does not contain the attribute (or that many instances of it), returns 0 instead. If *pbMore* is nonzero, the **getBytes** method sets it to TRUE if there are more instances of the attribute after the one returned, otherwise to FALSE. Use this to determine whether to make another call to **getBytes** to retrieve other instances of the attribute.	
getBytesByType	**const abytes_t ****pDict***-> getBytesByType(dex_AttributeDictionary_t ****pDict***,const abytes_t ****pszAttribute***, int** *iIndex***, abool_t ****pbMore* **)**
Returns the value of the *iIndex* ed instance of the attribute from the dictionary as a sequence of bytes. If the dictionary does not contain the attribute (or that many instances of it), 0 is returned instead. If *pbMore* is nonzero, sets the variable pointed to TRUE if there are more instances of the attribute after the one returned, otherwise to FALSE. Use this to determine whether to make another call to **get** to retrieve other instances of the attribute.	
getByType	**const char ****pDict***->getByType(dex_AttributeDictionary_t ****pDict***, const abytes_t ****pszAttribute***, int** *iIndex***, abool_t ****pbMore* **)**
Returns the value of the *iIndex* ed instance of the attribute from the dictionary, represented as a string. If the dictionary does not contain the attribute (or that many instances of it), returns the empty string instead. If *pbMore* is nonzero, the **getByType** method sets *pbMore* to TRUE if there are more instances of the attribute after the one returned, otherwise to FALSE. Use this to determine whether to make another call to **getByType** to retrieve other instances.	
getOption	**const char * getOption(dex_AttributeDictionary_t ****pDict***, int** *iArgType***, ...* **)**
Gets the data for an option as a string. Note that this function always returns a pointer to a string, which can be zero length if the option does not exist or has length zero. To find out if the option exists, use **getOptionBytes** or specify DEX_INSTANCE_COUNT.	
getOptionBytes	**const abytes_t * getOptionBytes(dex_AttributeDictionary_t ****pDict***, int** *iArgType***, ...* **)**

Method	Syntax
Gets the data for an option as a sequence of bytes. Note that this function returns a null pointer if the option does not exist, and an **abytes_t** with a zero-length buffer if the option exists but is zero bytes long.	
getType	**const abytes_t *** *pDict*->**getType(dex_AttributeDictionary_t*** *pDict*, **const char*** *pszAttribute* **)**
Returns a pointer to the byte sequence defining the attribute, if the attribute name matches a configured attribute, otherwise 0.	
isValidisV4isV6	**abool_t isValid(dex_AttributeDictionary_t ****pDict* **)abool_t isV4(dex_AttributeDictionary_t ****pDict* **)abool_t isV6(dex_AttributeDictionary_t ****pDict* **)**
The **isValid** method returns TRUE if there is a request or response (depending on the dictionary passed in); FALSE otherwise. Extensions such as **lease-state-change** can use this method to determine whether a dictionary is available. The **isV4** method returns TRUE if this extension is being called for a DHCPv4 packet; FALSE otherwise. Calling this method from an **init-entry** routine returns FALSE. The **isV6** method returns TRUE if this extension is being called for a DHCPv6 packet; FALSE otherwise. Calling this method from an **init-entry** routine returns FALSE.	
log	**abool_t** *pDict*->**log(dex_AttributeDictionary_t ****pDict*, **int** *eLevel*, **const char ****pszFormat*, ... **)**
Returns a message in the DHCP server logging system. The *eLevel* should be one of DEX_LOG_ERROR, DEX_LOG_WARNING, or DEX_LOG_INFO. The *pszFormat* is treated as a printf style format string, and it, along with the remaining arguments, are formatted and sent to the logging system at the specified level. **Note** Use the DEX_LOG_ERROR and DEX_LOG_WARNING levels sparingly, because the server flushes its log file with messages logged at these levels. Using these levels for messages that are likely to occur frequently (such as client requests) can have severe impact on disk I/O performance.	
moveToOption	**abool_t moveToOption(dex_AttributeDictionary_t ****pDict*, **int** *iArgType*, ... **)**
Sets the context for subsequent **get**, **put**, and **remove** option operations. Note that the context can become invalid if the option is removed (such as with **removeOption**).	
put	**abool_t** *pDict*->**put(dex_AttributeDictionary_t ****pDict*, **const char ****pszAttribute*, **const char ****pszValue*, **int** *iIndex* **)**
Converts *pszValue* to a sequence of bytes, according to the definition of *pszAttribute* in the server configuration. Associates that sequence of bytes with the attribute in the dictionary. If *iIndex* is the special value DEX_REPLACE, replaces any existing instances of the attribute with a single value. If the special value DEX_APPEND, it appends a new instance of the attribute to its list. If the special value DEX_AUGMENT, puts the attribute only if there is not one already. Otherwise, inserts a new instance at the position indicated. Returns TRUE unless the attribute name does not match any configured attributes or the value could not be converted to a legal value.	

Method	Syntax
putBytes	**abool_t** *pDict*->**putBytes(dex_AttributeDictionary_t** ***pDict*, const char** ****pszAttribute*, const abytes_t** ****pszValue*, int** *iIndex* **)**
Associates *pszValue* with the *pszAttribute* in the dictionary. If *iIndex* is the special value DEX_REPLACE, replaces any existing instances of the attribute with a single new value. If the special value DEX_APPEND, appends a new instance of the attribute to its list. If the special value DEX_AUGMENT, puts the attribute only if there is not one already. Otherwise, inserts a new instance at the position indicated. Returns TRUE unless the attribute name does not match a configured one.	
putBytesByType	**abool_t** *pDict*->**putBytesByType(dex_AttributeDictionary_t** ***pDict*, const abytes_t** ****pszAttribute*, const abytes_t** ****pszValue*, int** *iIndex* **)**
Associates *pszValue* with the *pszAttribute* in the dictionary. If *iIndex* is the special value DEX_REPLACE, replaces any existing instances of the attribute with the new value. If the special value DEX_APPEND, appends a new instance of the attribute to its list. If the special value DEX_AUGMENT, puts the attribute only if there is not one already. Otherwise, inserts a new instance of the attribute at the position indicated.	
putByType	**abool_t** *pDict*->**putByType(dex_AttributeDictionary_t** ***pDict*, const abytes_t** ****pszAttribute*, const char** ****pszValue*, int** *iIndex* **)**
Converts *pszValue* to a sequence of bytes, according to the definition of *pszAttribute* in the server configuration. Associates that sequence of bytes with the attribute in the dictionary. If *iIndex* is the special value DEX_REPLACE, replaces any existing instances of the attribute with a single new value. If the special value DEX_APPEND, appends a new instance of the attribute to its list. If the special value DEX_AUGMENT, puts the attribute only if there is not one already. Otherwise, inserts a new instance at the position indicated.	
putOption	**abool_t putOption(dex_AttributeDictionary_t** ***pDict*, const char** ****pszValue*, int** *iArgType*, ... **)**
Adds an option and its data or modifies the data for an option	
putOptionBytes	**abool_t putOptionBytes(dex_AttributeDictionary_t** ***pDict*, const abytes_t** ****pValue*, int** *iArgType*, ... **)**
Adds an option and its data or modifies the data for an option.	
remove	**abool_t** *pDict*->**remove(dex_AttributeDictionary_t** ***pDict*, const char** ****pszAttribute*, int** *iIndex* **)**
Removes the attribute from the dictionary. If *iIndex* is the special value DEX_REMOVE_ALL, removes any existing instances of the attribute. Otherwise, removes the instance at the position indicated. Returns TRUE, even if the dictionary did not contain that attribute at that index, unless the attribute name does not match any configured one.	

Method	Syntax
removeByType	**abool_t** *pDict*->**removeByType(dex_AttributeDictionary_t** ******pDict*, **const abytes_t** ******pszAttribute*, **int** *iIndex* **)**
Removes the attribute from the dictionary. If *iIndex* is the value DEX_REMOVE_ALL, removes any existing instances of the attribute. Otherwise, removes the instance at the position indicated. Always returns TRUE, even if the dictionary does not contain that attribute at that index.	
removeOption	**abool_t removeOption(dex_AttributeDictionary** ******pDict*, **int** *iArgType*, ... **)**
Removes an option. Note that if you omit DEX_INDEX, a DEX_INDEX of DEX_REMOVE_ALL is assumed (this removes the entire option).	
setObject	**abool_t setObject(dex_AttributeDictionary_t** ******pDict* , **int** *iObjectType*, **int** *iObjArgType*, ...)
Sets the object for **get**, **put**, and **remove** methods, and alters the message on which the new option methods operate. DHCPv6 extensions primarily use this method to access the leases and prefixes available for the client and link, or to get message header fields or options from relay packets. Unlike in DHCPv4, where one lease and scope are associated with a response, a DHCPv6 response can involve several leases and prefixes. Returns TRUE if the object exists; FALSE otherwise. For sample usage, see the Handling Object Data, on page 53. **Note** For leases not associated with the current client, only minimal information is available.	
trace	**abool_t** *pDict*->**trace(dex_AttributeDictionary_t** ******pDict*, **int** *iLevel*, **const char** ******pszFormat*, ... **)**
Returns a message in the DHCP server packet tracing system. At level 0, no tracing occurs. At level 1, it traces only that the server received the packet and sent a reply. At level 4, it traces everything. The remaining arguments are concatenated and sent to the tracing system at the specified level. The default tracing is set using the DHCP server *extension-trace-level* attribute.	

DEX Environment Dictionary Methods

The environment dictionary uses active commands, called *methods* , with which you can change and access the dictionary values. The table below lists the methods to use with the environment dictionary. In this case, you can define the *pDict* variable as **pEnviron**, as in:

```
pEnviron->log( pEnviron, DEX_LOG_INFO, "Environment hello world" );
```

Table 11: DEX Environment Dictionary Methods

Method	Syntax
allocateMemory	**void** ******pDict*->**allocateMemory(dex_EnvironmentDictionary_t** ******pDict*, **unsigned int** *iSize* **)**
Allocates memory for extensions that persists only for the lifetime of this request.	

Method	Syntax
clear	**void** *pDict*->**clear(dex_EnvironmentDictionary_t** ***pDict** **)**
Removes all entries from the dictionary.	
containsKey	**abool_t** *pDict*->**containsKey(dex_EnvironmentDictionary_t** ***pDict*, const char** ***pszKey** **)**
Returns TRUE if the dictionary contains the key, otherwise FALSE.	
firstKey	**const char** ***pDict*->**firstKey(dex_EnvironmentDictionary_t** ***pDict** **)**
Returns the name of the first key in the dictionary. Note that the keys are not stored sorted by name. If a key does not exist, returns zero.	
get	**const char** ***pDict*->**get(dex_EnvironmentDictionary_t** ***pDict*, const char** ***pszKey** **)**
Returns the value of the key from the dictionary. If a key does not exist, returns the empty string.	
isEmpty	**abool_t** *pDict*->**isEmpty(dex_EnvironmentDictionary_t** ***pDict** **)**
Returns TRUE if the dictionary has 0 entries, otherwise FALSE.	
log	**abool_t** *pDict*->**log(dex_EnvironmentDictionary_t** ***pDict*, int** *eLevel*, **const char** ***pszFormat*, ...** **)**
Returns a message in the DHCP server logging system. The *eLevel* should be one of DEX_LOG_ERROR, DEX_LOG_WARNING, or DEX_LOG_INFO. The *pszFormat* is treated as a printf style format string, and it, along with the remaining arguments, are formatted and sent to the logging system at the specified level. **Note** Use the DEX_LOG_ERROR and DEX_LOG_WARNING levels sparingly, because the server flushes its log file with messages logged at these levels. Using these levels for messages that are likely to occur frequently (such as client requests) can have severe impact on disk I/O performance.	
nextKey	**const char** ***pDict*->**nextKey(dex_EnvironmentDictionary_t** ***pDict** **)**
Returns the name of the next key in the dictionary that follows the key returned in the last call to **firstKey** or **nextKey**. If a key does not exist, returns zero.	
put	**abool_t** *pDict*->**put(dex_EnvironmentDictionary_t** ***pDict*, const char** ***pszKey*, const char*** *pszValue* **)**
Associates a value with the key, replacing an existing instance of the key with the new value.	
remove	**abool_t** *pDict*->**remove(dex_EnvironmentDictionary_t** ***pDict*, const char** ***pszKey** **)**

Method	Syntax
Removes the key and the associated value from the dictionary. Always returns TRUE, even if the dictionary did not contain the key.	
size	**int** *pDict*->**size(dex_EnvironmentDictionary_t** ***pDict*)**
Returns the number of entries in the dictionary.	
trace	**abool_t** *pDict*->**trace(dex_EnvironmentDictionary_t** ****pDict*, int** *iLevel*, **const char** ****pszFormat*, ...)**
Returns a message in the DHCP server packet tracing system. At level 0, no tracing occurs. At level 1, it traces only that the server received the packet and sent a reply. At level 4, it traces everything. The remaining arguments are concatenated and sent to the tracing system at the specified level. The default tracing is set using the DHCP server *extension-trace-level* attribute.	

Differences in get, put, Option, Bytes, and OptionBytes Methods

There are differences among the following DEX extension methods:

- **get** and **put**
- **getOption** and **putOption**
- **getBytes** and **putBytes**
- **getOptionBytes** and **putOptionBytes**

The **get** and **getOption** methods return the requested information formatted as a string. The server converts the data to the string depending on the expected data type for the dictionary item. If the data type is unknown, the server returns the data in blob string format.

The **getBytes** and **getOptionBytes** methods return the requested information as the raw bytes (a pointer to a buffer and the size of that buffer). The server should have to read this buffer only, and it contains only the data from the option (no null terminator has been added, for example).

The **put** and **putOption** methods expect the data to be written as a formatted string. The server converts the data from the string depending on the expected data type for the dictionary item. If the data type is unknown, it is expected to be in blob string format.

The server passes raw bytes to the **putBytes** and **putOptionBytes** methods (a pointer to a buffer and the size of that buffer). The server only reads these bytes.

Differences in get, put, remove, and ByType Methods

There are differences among the following DEX extension methods:

- **get**, **put**, and **remove**
- **getByType**, **putByType**, and **removeByType**

The server passes the **get**, **put**, and **remove** methods the name of the desired data item as a string. This requires that the server map the string to its internal data tables.

The server passes the **getByType**, **putByType**, and **removeByType** methods an internal data table reference, which the server must have previously obtained (such as in the extension init-entry) by calling the **getType**

method on the string. This speeds processing for extensions, which can be important in applications requiring high performance.

Note The internal data table that the **getType** method references is the same whether requested for the Request or Response dictionary. There is no need for separate **getType** calls on each dictionary for the same data item name.

Table 12: DEX iObjectType Values

i Object Type	Description
General definition: Object for which the context is to be changed.	
DEX_LEASE	Changes the lease (and prefix) context. Response dictionary only. Allows *iObjTypeArg* : DEX_BY_IPV6ADDRESS DEX_BY_IPV6PREFIX DEX_BY_INSTANCE DEX_INITIAL
DEX_MESSAGE	Changes the message context to a relay message or the client message. Request and response dictionaries. Allows *iObjArgType* : DEX_INITIAL DEX_RELAY DEX_BY_NUMBER
DEX_PREFIX	Changes the prefix context, but does not change the lease context. Response dictionary only. Allows *iObjTypeArg* : DEX_BY_IPV6ADDRESS DEX_BY_IPV6PREFIX DEX_BY_INSTANCE DEX_BY_NAME DEX_INITIAL

Table 13: DEX iObjArgType Values

iObjArgType	Description
General definition: By what means the context is to be changed.	

iObjArgType	Description
DEX_BY_INSTANCE	Used with DEX_LEASE or DEX_PREFIX *iObjectType* . Requires that **int** follows to specify the instance number (starting with 0). Used to walk through the list of all available objects, but only through the list of objects applicable to the current request or response: for DEX_LEASE, the leases for that client (if any); for DEX_PREFIX, the prefixes on the current link (if any). Used with DEX_MESSAGE, a synonym for DEX_RELAY.
DEX_BY_IPV6ADDRESS	Used with DEX_LEASE and DEX_PREFIX *iObjectType* only. Requires that **const unsigned char *** follows to specify the 16-byte address.
DEX_BY_IPV6PREFIX	Used with DEX_LEASE or DEX_PREFIX *iObjectType* . Requires that **const unsigned char *** follows to specify a 17-byte prefix buffer (16-byte address followed by a 1-byte prefix length).
DEX_BY_NAME	Used with the DEX_PREFIX *iObjectType* only. Requires that a **const char *** follows to specify the name of the desired object.
DEX_INITIAL	Resets the context back to the original for the request or response, and has no additional argument. Sets the lease and prefix (DEX_LEASE), prefix (DEX_PREFIX), or message (DEX_MESSAGE) to what it was when the extension was originally called (which can be none).
DEX_RELAY	Used with DEX_MESSAGE *iObjectType* only. Requires that **int** follows to specify the relay (0 specifies the relay closest to the client). To set the message context back to the client, use **setObject(pDict, DEX_MESSAGE, DEX_INITIAL)**.

iArg Type	Description
General definition: Action and argument that follows the context. There can be any number of *iArgType* instances in the calls.	

iArg Type	Description
DEX_ARG_ARRAY	Requires that a pointer to an array of **dex_OptionsArgs_t** follow, and is an alternative to specifying the argument list. Each **dex_OptionsArgs_t** structure has two fields: • *iArgType* —One of the *iArgType* DEX values in this table. • *pData* —Data (integer), pointer to the data (for strings and other data types), or ignored (if the *iArgType* takes no arguments). Note that once the server encounters the DEX_ARG_ARRAY (in an argument list or in an array of **dex_OptionsArgs_t**), it ignores any subsequent arguments in the original list.
DEX_END	**Note** Required, has no additional argument, and marks the end of the argument list.
DEX_ENTERPRISE_NAME	Requires that **const char *** follow to specify the option definition set name, from which the server extracts the enterprise-id to get the vendor option data. Valid only for vendor-identifying options. Requires that the vendor option definition set exists.
DEX_ENTERPRISE_ID	Requires that **int** follow to specify the enterprise-id for the vendor.
DEX_HOME	Moves the context back to the client or relay message options. Has no additional argument. Always returns success. If used, must be the first *iArgType* . Valid only for **getOption**, **getOptionBytes**, and **moveToOption** methods.

iArg Type	Description
DEX_INDEX	Requires that **int** follow with the index of the option data (if any array of data is to be acted on). If omitted, index 0 is assumed, except for **removeOption**, in which case DEX_REMOVE_ALL is assumed. Use the special value DEX_RAW to get, put, or remove the entire option data. However, for the DHCPv4 Vendor-Identifying Vendor Options (RFC 3925 and RFC 4243), DEX_RAW returns the data for only one vendor (based on the instance or enterprise-id) and not that for the entire option. The DEX_RAW special value accesses the entire option (or suboption) data. It provides consistent access to the data, regardless of what the option definitions might specify in terms of the data type and repeat counts of the data type. It is recommended for general-purpose extensions that decode the data. Use the special values DEX_REPLACE (replace a value), DEX_APPEND (add to end), and DEX_AUGMENT (add if no value currently exists) with **putOption** and **putOptionBytes** methods, which operate the same as the **put**, **putByType**, **putBytes**, and **putBytesByType** methods. Use DEX_REMOVE_ALL for **removeOption** to remove the option completely.
DEX_INDEX_COUNT	Results in an **int** value returned with the count of the number of indexed entries of the option, rather than the option data. Has no additional argument, and cannot be used with DEX_INDEX or DEX_INSTANCE_COUNT. DEX_END must follow. Valid only for **getOption** and **getOptionBytes**.
DEX_INSTANCE	Requires that **int** follow to specify the instance of the option (valid only for DHCPv6 options, which can have more than one instance). 0 specifies the first instance.
DEX_INSTANCE_COUNT	Results in an **int** value returned with the count of the number of instances of the option, rather than the option data. Has no additional argument and cannot be used with DEX_INSTANCE. DEX_END must follow. Valid only for **getOption** and **getOptionBytes**.
DEX_MORE	Requires that **abool_t *** follow to specify the location at which a **more** flag is to be written. This location is set to TRUE if more array items exist beyond the index that DEX_INDEX specified. Valid only for **getOption** and **getOptionBytes** methods.

iArg Type	Description
DEX_MOVE_TO	Leaves the context at the option or suboption immediately preceding DEX_MOVE_TO. Has no additional argument. If omitted, the context does not change. Use **moveToOption** to move the context without getting any data. Valid only for **getOption** and **getOptionBytes** methods. **Note** An attempt to move to an option or suboption that does not exist logs an error. Use **moveToOption** if your extension did not previously confirm that the option exists.
DEX_OPTION_NAME	Requires that **const char *** follow to specify the desired option name. Option names should be in the **dhcpv4-config** or **dhcpv6-config** option definition set.
DEX_OPTION_NUMBER	Requires that **const char *** follow to specify the desired option name. Option names should be in the **dhcpv4-config** or **dhcpv6-config** option definition set.
DEX_PARENT	Moves the context to the parent option. Has no additional argument. It does not move beyond the client or relay message and returns FALSE if the context does not change. If used, must be the first *iArgType* . Valid only for **getOption**, **getOptionBytes**, and **moveToOption** methods.
DEX_SUBOPTION_NAME	Requires that **const char *** follow to specify the name of the desired suboption. Suboptions must be in the current option definition.
DEX_SUBOPTION_NUMBER	Requires that **int** follow to specify the desired suboption number. Suboption numbers should be in the current option definition, although there is no requirement that a definition exists. However, if the suboption does not exist, it is assumed to be a byte blob of data.
DEX_VENDOR_NAME	Requires that **const char *** follow to specify the vendor string. The string serves only to find the appropriate option definition set.

Handling Objects and Options

The following sections describe specialized ways of handling DHCP objects and options in extensions.

Using Object and Option Handling Methods

Extensions can call methods to set DHCP objects, and get, move to, put, and remove DHCP options. The methods are **setObject**, **getOption**, **moveToOption**, **putOption**, and **removeOption** methods in Tcl and C/C++.

These new callback methods were introduced primarily to provide support for DHCPv6. However, you can use the option-related functions for DHCPv4. In fact, it is recommended to use these methods for DHCPv4, because they provide richer access to options than the original **get**[**Bytes**], **get**[**Bytes**]**ByType**, **put**[**Bytes**], **put**[**Bytes**]**ByType**, and **remove**[**ByType**] methods.

Tip See DEX Request and Response Dictionary Methods, on page 39 for the different usages of some of these methods in C/C++.

For DHCPv6, you must use the **setObject**, **getOption**, **moveToOption**, **putOption**, and **removeOption** methods to access options. The **setObject** method was introduced for DHCPv6, because there can be many leases, prefixes, and messages (client or multiple relay) that an extension might want to access. So, **setObject** serves to set the context for subsequent calls to get request and response dictionary data items and options. When the server calls an extension, the context is set to the current lease (if applicable), prefix (if applicable), and client message. For example, when the server calls the **pre-packet-encode** extension point, only the request and response dictionary message context is valid, and set to the corresponding client message, because there is no lease or prefix associated with this extension point. However, when the server calls the **lease-state-change** extension point, it sets the response dictionary lease context to the lease on which the state has changed, sets the response dictionary prefix context to the prefix for the lease, and sets the request and response dictionary message context to the corresponding client message.

Options and Suboptions in C/C++

Some C/C++ extensions provide specialized argument type values to handle DHCP options and suboptions. The DEX_OPTION_* argument type specifies to use the standard DHCPv4 or DHCPv6 option definition set and not the definitions under an option (or suboption). So, DEX_OPTION_* means that the server looks up the option name or number in the standard DHCPv4 or DHCPv6 option definition set, whereas DEX_SUBOPTION_* means that the server looks up the suboption name or number of the current option definition (if any).

Thus, when you access options in DHCPv6, you often use DEX_OPTION_* followed by DEX_OPTION_* when options are encapsulated. You would use DEX_SUBOPTION when looking at vendor options. For DHCPv4, you would use DEX_OPTION at the client packet level, and then DEX_SUBOPTION perhaps one or more times, depending on the nesting level. Generally, only options have enterprise numbers or vendor names, but there is no prohibition on this. The option definition sets determine what is valid (although one can walk off definitions, at which point everything is treated as binary bytes and thus it limits what is possible, and you cannot use the option or suboption names, but must use numbers).

The option ordering rules for the **getOption**, **moveToOption**, **putOption**, and **removeOption** methods are similar to the **request** expression syntax. The ordering generally consists of:

- Preamble clause ([**parent** | **home**])
- Option clause (**option** [**vendor** | **enterprise**] [**instance**])
- Suboption clause (**suboption** [**vendor** | **enterprise**] [**instance**])
- End clause ([**instance-count** | **index-count** | [**index**] [**more**] **end**)

You can construct calls by using a preamble clause, followed by zero or more option clauses, followed by zero or more suboption clauses (which may themselves be followed by option and suboption clauses), followed by an end clause. Note that some things are possible only through a **get** method (such as **instance-count**, **index-count**, and **more**), and **move-to** can appear anywhere to move the context to the current option or suboption.

The option definition determines its data format, which can differ from what the older functions return for a specific option. To handle specific options:

- For the vendor class options (*v-i-vendor-class* [124] for DHCPv4 and *vendor-class* [16] for DHCPv6), if you ask for a specific instance of the option (instead of by enterprise-id or name), the only way to get the enterprise-id is to ask for the raw data (DEX_INDEX with DEX_RAW).
- For the DHCPv4 vendor options (*v-i-vendor-class* [124] and *v-i-vendor-opts* [125]), operating on the raw data (DEX_INDEX with DEX_RAW) only applies to an instance (preset value 0) of that option, not the entire option. There is no way to get the entire data for this option, which means that you cannot use **putOption** for the entire data. This is not an issue with the DHCPv6 vendor options, because these are separate options.
- If one of the DHCPv4 vendor options (124 or 125) is not formatted properly, the entire data is returned as a blob (if you asked for instance 0 and did not specify a particular enterprise-id). However, if an extension tries to use **putOption**, depending on the operation, that data might be appended to the existing data, and the result will be formatted incorrectly.
- For the vendor options, if there is no option, **putOption(pDict, "01:02", DEX_OPTION_NUMBER, 124, DEX_END)** fails because no enterprise-id is available. However, **putOption(pDict, "00:00:00:09:04:03:65:66:67", DEX_OPTION_NUMBER, 124, DEX_END)** will work because it is assumed that 00:00:00:09 is the enterprise-id and the bytes following it starting with 04 are the length of the option data of that enterprise-id. Note that the length byte is validated in this case, and **putOption** fails if it does not have the correct length. The recommended way to add this data is to use **putOption(pDict, "65:66:67", DEX_OPTION_NUMBER, 124, DEX_ENTERPRISE_ID, 9, DEX_END)**.

Examples of Option and Object Method Calls

These sections include some examples of how to use methods to handle DHCP option and object data.

Handling Vendor Class Option Data

For DHCPv4, to include the Vendor-Identifying Vendor Class option (124) data for two enterprise-ids in the response to the client, here is some sample Tcl code that uses the **putOption** method:

```
$response putOption 65:66:67 option 124 enterprise 999998
 #adds "abc" (65:66:67) under enterprise-id 999998
$response putOption 68:69:6a:6b option v-i-vendor-class enterprise 999998 index append
 #appends "defg" (68:69:6a:6b) under the same enterprise-id
$response putOption 01:02:03:04 option 124 enterprise 999999
 #adds 01:02:03:04 under enterprise-id 999999
```

To get the options, use the **getOption** method:

```
$response getOption option v-i-vendor-class instance-count
 #returns 2 because there were two instances added (enterprise id 999998 and enterprise id
 999999)
$response getOption option 124
 #returns index 0 of instance 0, which is 65:66:67
```

```
$response getOption option 124 index-count
 #returns 2 because there were two vendor classes added for the first enterprise id (9999998)
$response getOption option 124 index raw
 #returns 00:0f:42:3e:09:03:65:66:67:04:68:69:6a:6b for the complete encoding of the
enterprise-id 999998 data (see RFC 3925)
$response getOption option 124 index 1
 #returns 68:69:6a:6b
$response getOption option 124 instance 1 index-count
 #returns 1 because there is only one vendor class
$response getOption option 124 instance 1 index raw
 #returns 00:0f:42:3f:05:04:01:02:03:04 for the complete encoding of the enterprise-id
999999 data (see RFC 3925)
$response getOption option 124 enterprise 999999
 #returns 01:02:03:04
```

To remove the data, two **removeOption** calls are necessary because of the two separate enterprise-ids:

```
$response removeOption option 124
$response removeOption option 124
```

Handling Object Data

Suppose that at the **pre-packet-encode** extension point you want to extract data for all of the leases for the client. Here is sample Tcl code that uses the **setObject** method:

```
proc logleasesinit { request response environ } {
   if { [$environ get "extension-point"] == "initialize" } {
     # Set up for DHCPv6 only]
     $environ put dhcp-support "v6"
     $environ put extension-extensionapi-version 2
  }
}
proc logleases { request response environ } {
   for { set i 0 } { 1 } { incr i } {
     # Set context to next lease
     if { ![$response setObject lease $i] } {
     # Lease does not exist, so done
     break
  }
  # Log the lease address, prefix name, and prefix address
  $environ log LOG_INFO "Lease [$response get lease-ipaddress], Prefix\
     [$response get lease-prefix-name] - [$response get prefix-address]"
  }
  # Restore the lease context to where we started
    $response setObject lease initial
    # Do other things...
}
```

The C++ equivalent code for this might be:

```
// Print the current leases for the client
for( int i=0; ; i++ ) {
   if( !pRes->setObject( pRes, DEX_LEASE, DEX_BY_INSTANCE, i ) )
      break;
   const char *pszLeaseAddress =
      pRes->get( pRes, "lease-ipaddress", 0, 0 );
   if( pszLeaseAddress == 0 )
      pszLeaseAddress = "<error>";
   const char *pszPrefixName =
```

DHCP Extension Dictionary

```
      pRes->get( pRes, "prefix-name", 0, 0 );
   if( pszPrefixName == 0 )
      pszPrefixName = "<error>";
   pEnv->log(pEnv, DEX_LOG_INFO,
     "Lease %s, Prefix %s",
   pszLeaseAddress, pszPrefixName );
}
```