Oracle® Solaris DHCP Service Developer's Guide

Beta

ORACLE®

Contents

Figures

Tables

Preface

This *Oracle Solaris DHCP Service Developer's Guide* provides information for developers who want to use a data storage facility not currently supported by the Solaris™ DHCP service. The manual gives an overview of the data access framework used by Solaris DHCP, general guidelines for developers, and a listing of the API functions you use to write a module to support the new data store.

Who Should Use This Book

This book is intended for Solaris programmers interested in extending the data storage choices available to the Solaris DHCP service.

How This Book Is Organized

This book consists of the following chapters:

Chapter 1, "Overview of Solaris DHCP Data Access Architecture," provides an overview of the architecture used for data access in the DHCP service.

Chapter 2, "Architecture Features for Module Writers," discusses what the architecture requires of you.

Chapter 3, "Service Provider Layer API," describes the API functions you will export.

Chapter 4, "Code Samples and Testing," provides sample code templates and pointers to locations on Sun's web site where you can find additional aids for writing and debugging code for the public module.

Documentation, Support, and Training

See the following web sites for additional resources:

- Documentation (http://docs.sun.com)
- Support (http://www.oracle.com/us/support/systems/index.html)
- Training (http://education.oracle.com) – Click the Sun link in the left navigation bar.

Oracle Software Resources

Oracle Technology Network (http://www.oracle.com/technetwork/index.html) offers a range of resources related to Oracle software:

- Discuss technical problems and solutions on the Discussion Forums (http://forums.oracle.com).
- Get hands-on step-by-step tutorials with Oracle By Example (http://www.oracle.com/technetwork/tutorials/index.html).
- Download Sample Code (http://www.oracle.com/technology/sample_code/index.html).

Typographic Conventions

The following table describes the typographic conventions that are used in this book.

TABLE P–1 Typographic Conventions

Typeface	Meaning	Example
AaBbCc123	The names of commands, files, and directories, and onscreen computer output	Edit your .login file.
		Use ls -a to list all files.
		machine_name% you have mail.
AaBbCc123	What you type, contrasted with onscreen computer output	machine_name% **su**
		Password:
aabbcc123	Placeholder: replace with a real name or value	The command to remove a file is rm *filename*.
AaBbCc123	Book titles, new terms, and terms to be emphasized	Read Chapter 6 in the *User's Guide*.
		A *cache* is a copy that is stored locally.
		Do *not* save the file.
		Note: Some emphasized items appear bold online.

Shell Prompts in Command Examples

The following table shows the default UNIX system prompt and superuser prompt for shells that are included in the Oracle Solaris OS. Note that the default system prompt that is displayed in command examples varies, depending on the Oracle Solaris release.

TABLE P–2 Shell Prompts

Shell	Prompt
Bash shell, Korn shell, and Bourne shell	$
Bash shell, Korn shell, and Bourne shell for superuser	#
C shell	machine_name%
C shell for superuser	machine_name#

1
◆ ◆ ◆ **C H A P T E R 1**

Overview of Solaris DHCP Data Access Architecture

This chapter presents an overview of the architecture of the Solaris Dynamic Host Configuration Protocol (DHCP) service introduced in the Solaris 8 7/01 operating environment. This overview can help you see where your work will fit into the architecture.

For general information about the Solaris DHCP service, see Chapter 12, "About Solaris DHCP (Overview)," in *System Administration Guide: IP Services*.

The following topics are included:

Modular Framework

The Solaris DHCP service includes the DHCP daemon, administrative tools, and separate data access modules (called *public modules*) for different data storage facilities. Solaris DHCP provides an API that enables you to write your own public modules, implemented as shared objects, to support any data storage facility you want. When you integrate your public module into the Solaris DHCP framework, the DHCP service stores its data in your database using your public module. Public modules can be delivered independently of the Solaris DHCP service, enabling anyone to develop and deliver modules to support any data storage facility.

The first release of Solaris DHCP using this architecture provides public modules for ASCII files, NIS+, and file-system-based binary data stores. This manual provides information that enables developers to create their own public modules for any database.

DHCP Server Multithreading

The DHCP server implements multithreading, enabling it to service many clients simultaneously. Public modules are required to be MT-SAFE to support multithreading by the DHCP server, and this in itself allows the DHCP service to handle a larger number of clients. However, the capacity of the DHCP server is largely dependent on the capabilities of the data storage facility and the efficiency of the public module used to access the data. You can potentially increase the performance and capacity of your Solaris DHCP service by creating a public module for using a fast, high-capacity data storage facility.

Data Access Layers

The Solaris DHCP modular framework implementation employs the following data access layers:

- Application/Service Layer, consisting of all consumers of DHCP service data such as the DHCP daemon (`in.dhcpd`), command line management utilities (`pntadm`, `dhtadm`, `dhcpconfig`), `dhcpmgr`, and report generators. These data consumers interface with the DHCP service using calls to API functions implemented by the Framework Configuration Layer of the architecture.

- Framework Configuration Layer, consisting of the shared library `libdhcpsvc.so` and the `/etc/inet/dhcpsvc.conf` configuration file. The Framework Configuration Layer connects the Application/Service Layer and the Service Provider Layer. See "The Framework Configuration Layer" on page 15 for more information about the Framework Configuration Layer.

- Service Provider Layer, consisting of public modules that implement the Service Provider API functions, which are used by the Application/Service Layer through the Framework Configuration Layer to manipulate the data store containers and the records within them. The data store containers are the `dhcptab` and DHCP network tables.

The following figure shows the interaction of the architecture layers.

FIGURE 1–1   Architecture of Data Store Access in DHCP Service
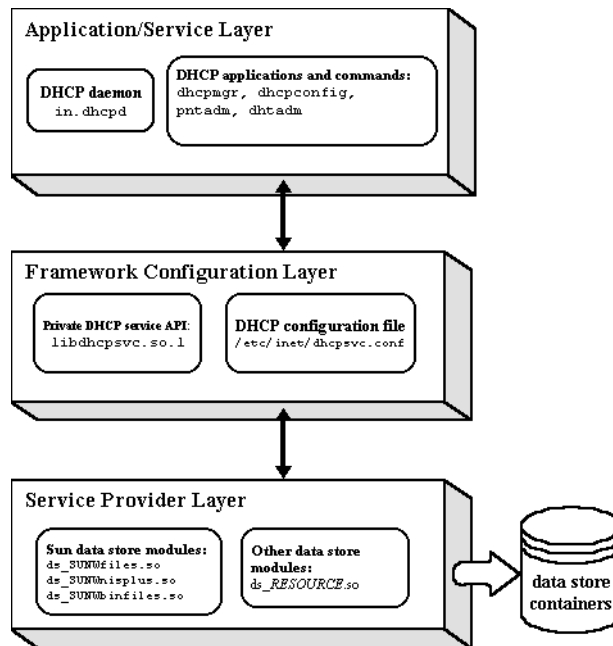
The Framework Configuration Layer

Functions implemented in libdhcpsvc.so are used by the Application/Service Layer to:

- locate, load, and unload public modules
- manage data container version changes
- access the data store containers
- manipulate data store records in the containers

The /etc/inet/dhcpsvc.conf contains a number of configuration parameters for the DHCP service, including the following keywords relevant to the public module developer:

RESOURCE   Public module to load. The value of RESOURCE matches the public module name. For example, the RESOURCE=SUNWfiles refers to public module ds_SUNWfiles.so. "Naming the Public Module and Data Store Containers" on page 22 explains the rules for naming public modules.

PATH   Location of DHCP containers within the data service that the public module exports. The value of PATH is specific to the data service. For example, a UNIX™ path name would be assigned to PATH for the SUNWfiles resource.

RESOURCE_CONFIG   Configuration information specific to the public module. This is an optional keyword that you can use if the data service requires configuration information, such as authentication from the user. If you use this keyword, you must provide a public module management bean to prompt the user for information to set the keyword value. See "Data Service Configuration and DHCP Management Tools" on page 25. The module must also export the configure() function to receive the value of this keyword during module load time. See "configure()" on page 29 for more information.

The Framework Configuration Layer also provides to the Service Provider Layer an optional API synchronization service, described in "Synchronizing Access to File-System-Based Containers" on page 20.

The Service Provider Layer API

The Service Provider Layer API consists of functions, data structures, and manifest constants contained in the /usr/include/dhcp_svc_public.h file.

The functions are summarized in the following table, with links to sections with more detail about each function.

TABLE 1–1   Service Provider Layer API Functions

API Function	Use
General functions for all data store containers	
"configure()" on page 29	Pass a configuration string to the data store. Optional function.
"mklocation()" on page 30	Create the location in which the data store will reside.
"status()" on page 31	Return general status information for the data store.
"version()" on page 31	Return the version of the Service Provider Layer API implemented by the data store container.
Functions for dhcptab containers	
"list_dt()" on page 32	Return the dhcptab container name.
"open_dt()" on page 32	Open or create the dhcptab container.
"lookup_dt()" on page 33	Perform a query for records in the dhcptab container.
"add_dt()" on page 34	Add a record to the dhcptab container.
"modify_dt()" on page 35	Modify an existing record in the dhcptab container.

TABLE 1–1 Service Provider Layer API Functions *(Continued)*

API Function	Use
"delete_dt()" on page 35	Delete a record from the dhcptab container.
"close_dt()" on page 36	Close the dhcptab container.
"remove_dt()" on page 36	Remove the dhcptab container from the data store.
Functions for DHCP network containers	
"list_dn()" on page 37	Return a list of DHCP network container names.
"open_dn()" on page 37	Open or create a DHCP network container.
"lookup_dn()" on page 38	Perform a query for records in a DHCP network container.
"add_dn()" on page 39	Add a record to a DHCP network container.
"modify_dn()" on page 39	Modify an existing record in a DHCP network container.
"delete_dn()" on page 40	Delete a record from a DHCP network container.
"close_dn()" on page 40	Close a DHCP network container.
"remove_dn()" on page 41	Remove a DHCP network container from the data store.

Data Store Containers

The dhcptab and DHCP network tables are referred to generically as data store *containers*. By default, Solaris DHCP provides support for the container formats shown in the following table.

Data Service Supported	Public Module
File-system-based, ASCII format	ds_SUNWfiles.so
NIS+ service	ds_SUNWnisplus.so
File-system-based, binary format	ds_SUNWbinfiles.so

2

Architecture Features for Module Writers

This chapter discusses architectural details you should keep in mind when creating a public module for a data service.

The following topics are included:

Function Categories

The Service Provider Layer API functions can be divided into three categories:

- Data store functions, which facilitate activities related to the public module and underlying data service themselves. These functions include `configure()`, `mklocation()`, `status()`, and `version()`.

- `dhcptab` container functions, which facilitate the creation of the `dhcptab` container, the writing of records to the `dhcptab` container, and the query of records in the `dhcptab` container. The `open_dt()` function creates a handle for the container, and the other functions take a pointer to that handle. The `close_dt()` function destroys the handle when it closes the container.

- Network container functions, which facilitate the creation of DHCP network containers, the writing of records to the network containers, and the query of records in the network containers. The `open_dn()` function creates a handle for the container, and the other functions take a pointer to that handle. The `close_dn()` function destroys the handle when it closes the container.

The functions are described in more detail in Chapter 3, "Service Provider Layer API."

Considerations for Multithreading

The DHCP server implements multithreading, which enables it to service many clients simultaneously. Public modules are required to be MT-SAFE to support multithreading by the DHCP server.

To make your module MT-SAFE, you must synchronize calls to `add_d?()`, `delete_d?()`, and `modify_d?()` so that they are called serially. For example, if one thread is inside `add_dn()` for a given DHCP network container, no other thread may be inside `add_dn()`, `delete_dn()`, `modify_dn()`, or `lookup_dn()` for that same container. If your public module supports a local file-system-based data service, you can use the synchronization service to take care of this for you. See "Synchronizing Access to File-System-Based Containers" on page 20 for more information.

Synchronizing Access to File-System-Based Containers

When you write a public module that provides access to containers in a local file-system-based data service (the data service runs on the same machine as the DHCP server), it can be difficult to synchronize access to the underlying data service between multiple processes and threads.

The Solaris DHCP synchronization service simplifies the design of public modules using local file-system-based data services by pushing synchronization up into the Framework Configuration Layer. When you design your module to use this framework, your code becomes simpler and your design cleaner.

The synchronization service provides public modules with per-container exclusive synchronization of all callers of the `add_d?()`, `delete_d?()`, and `modify_d?()` Service Provider Layer API calls. This means that if one thread is inside `add_dn()` for a given DHCP network container, no other thread may be inside `add_dn()`, `delete_dn()`, `modify_dn()` or `lookup_dn()` for that same container. However, other threads may be within routines that provide no synchronization guarantees, such as `close_dn()`.

Per-container shared synchronization of all callers of `lookup_d?()` is also provided. Thus, there may be many threads performing a lookup on the same container, but only one thread may perform an add, delete, or modify operation.

The synchronization service is implemented as a daemon (`/usr/lib/inet/dsvclockd`). Lock manager requests are made on the public module's behalf through Framework Configuration Layer API calls. The interface between the Framework Configuration layer and the lock manager daemon uses the Solaris doors interprocess communication mechanism. See, for example, `door_create` and `door_call`.

The Framework Configuration layer starts the dsvclockd daemon if a public module requests synchronization and the daemon is not already running. The daemon automatically exits if it manages no locks for 15 minutes. To change this interval, you can create a /etc/default/dsvclockd file and set the IDLE default to the number of idle minutes before the daemon terminates.

A public module notifies the Framework Configuration Layer that it requires synchronization services by providing the following global variable in one of the module's source files:

```
dsvc_synchtype_t dsvc_synchtype = DSVC_SYNCH_DSVCD;
```

A public module notifies the Framework Configuration Layer that it does *not* require synchronization services by including the following global variable in one of the module's source files:

```
dsvc_synchtype_t dsvc_synchtype = DSVC_SYNCH_NONE;
```

DSVC_SYNCH_DSVCD and DSVC_SYNCH_NONE are the only two synchronization types that exist currently.

Avoiding Update Collisions

The architecture provides a facility that helps a files-based module avoid record update collisions. The Service Provider API facilitates the maintenance of data consistency through the use of a per-record update signature, an unsigned 64–bit integer. The update signature is the d?_sig element of the d?_rec_t container record data structure, defined in /usr/include/dhcp_svc_public.h. All layers of the architecture use d?_rec_t records, from the Application/Service Layer through the Framework Configuration Layer API and on through to the Service Provider Layer API. Above the Service Provider Layer, the update signature is an opaque object which is not manipulated by users of the Framework Configuration Layer API.

When a module receives a d?_rec_t record through a Service Provider Layer API function call, it should perform a lookup in the data service to find a record that matches the key fields of the d?_rec_t, and compare the signature of the internal record against the d?_rec_t passed by the call. If the signature of the internal record does not match that of the passed record, then the record has been changed since the consumer acquired it from the public module. In this case, the module should return DSVC_COLLISION, which informs the caller that the record has been changed since it was acquired. If the signatures match, the module should increment the update signature of the argument record before it stores the record.

When a module receives a new d?_rec_t record through the Service Provider Layer API, the module must assign a value to the update signature before it adds the record to the container. The simplest way is to set the value to 1.

However, in certain rare situations a collision might not be detected if the signature always has the same initial value. Consider the following scenario. Thread A adds a record with a signature of 1, and Thread B looks up that record. Thread A deletes the record and creates a new record with the same key fields and a signature of 1 since it has just been created. Thread B then modifies the record it looked up, but that has already been deleted. The module compares the key fields and signatures of the record looked up by Thread B and the record in the data store, finds them to be the same, and makes the modification. Such a modification attempt should have been a collision because the records are, in fact, not the same.

The `ds_SUNWfiles.so` and `ds_SUNWbinfiles.so` modules provided with Solaris DHCP address such a possibility. They divide the update signature into two fields to ensure the uniqueness of each record's signature. The first 16 bit field of the update signature is set to a randomly generated number. This field never changes in the record after it is set. The lower 48 bit field of the signature is set to 1 and then incremented each time the record is updated.

Note – The modules provided with Solaris DHCP illustrate one approach you can use to avoid record update collisions. You can devise your own method or use a similar one.

Naming the Public Module and Data Store Containers

The public module and containers must both contain version numbers to enable the architecture's upgrading mechanism to work.

Public Module Name

You must use the following name format for your public module:

ds_*name*.so.*ver*

where *name* is the name of the module and *ver* is the container format version number. The *name* must use a prefix that is an internationally known identifier associated with your organization. For example, the public modules that Sun Microsystems provides have names prefixed with SUNW, the stock ticker symbol for Sun. For example, the NIS+ public module is named `ds_SUNWnisplus.so.1`. By including such an identifier in the module name, you avoid public module name collisions in the `/usr/lib/inet/dhcp/svc` public module directory.

If your company name is Inet DataBase, for example, you might call your module `ds_IDBtrees.so.1`

Container Name

The container names presented to the administrator through the administrative interface must always be `dhcptab` and the dotted IP network address for the DHCP network tables, such as `10.0.0.0`.

Internally, the data store container names must contain the version number to enable you to produce revisions of your container formats whenever necessary. This naming scheme allows the coexistence of multiple versions of a container, which is a requirement for the architecture's container version upgrade mechanism to work.

The names used for the containers should include a globally recognizable token to ensure that the names are unique.

For example, the NIS+ public module provided with Solaris DHCP would create the `dhcptab` container internally as `SUNWnisplus1_dhcptab`. The container for the 172.21.174.0 network table would be `SUNWnisplus1_172.21.174.0`.

If your company name is Inet DataBase, and your public module is `ds_IDBtrees.so.1`, you would name your containers `IDBtrees1_dhcptab` and `IDBtrees1_172.21.174.0`.

Container Record Formats

The Solaris DHCP service uses two types of DHCP containers: the `dhcptab` container and the DHCP network container.

The `dhcptab` container holds DHCP configuration data, described in the `dhcptab` man page. Only one instance of a `dhcptab` container is maintained in the DHCP service.

`dhcptab` records are passed between the Framework Configuration Layer and the Service Provider Layer by way of an internal structure, `dt_rec_t`. The include file `/usr/include/dhcp_svc_public.h` shows the structure.

Your public module must ensure that there are no duplicate `dhcptab` records. No two records can have identical key field values.

DHCP network containers contain IP address records, described in the `dhcp_network` man page. These containers are named to indicate the data store and the dotted IP address of the network to which the IP addresses belong, such as 10.0.0.0. Any number of DHCP network containers may exist, one for each network supported by the DHCP service.

DHCP network records are passed between the Framework Configuration Layer and the Service Provider Layer by way of an internal structure, `dn_rec_t`. The include file `/usr/include/dhcp_svc_public.h` shows the structure.

Your public module must ensure that there are no duplicate network container records. No two records can have identical key field values.

Passing Data Store Configuration Data

The Solaris DHCP data access architecture provides an optional feature for passing data-store-specific configuration data to the public module (and thus the data store). This feature is implemented as an ASCII string which is passed through the DHCP service management interface (dhcpconfig or dhcpmgr) and stored by the Framework Configuration Layer on the DHCP server machine. See the dhcpsvc.conf(4) man page for more information. You determine what kind of information is passed in the string, and the DHCP administrator provides the value of the string through the administration tool. The string might, for example, contain a user name and password needed to log in to a database.

To obtain the information from the DHCP administrator, you must write a JavaBeans™ component to present an appropriate dialog. The information is then passed to the management interface as a single ASCII string. You should document the format of the ASCII string token to facilitate debugging. To support this feature, the public module must implement and export the configure() function, described in Chapter 3, "Service Provider Layer API."

Note – The architecture does not encrypt the ASCII string. It is saved in clear text in the /etc/inet/dhcpsvc.conf file. If you require encrypted information, the bean must encrypt the information before passing it to the Framework Configuration Layer.

Upgrading Container Versions

You do not need to be concerned with container version upgrades, because the architecture facilitates the coexistence of different container versions when you follow the naming guidelines described in "Naming the Public Module and Data Store Containers" on page 22. The administrative tools use this feature of the architecture to enable DHCP administrators to automatically upgrade from one container version to another.

The container format version is set in the Framework Configuration Layer configuration file automatically, either by the installation (when upgrading Solaris DHCP) or through the administrative interface during initial DHCP service configuration. If you install a new version of a public module that includes a new container version, the administrative interface automatically detects the new version, and asks the administrator to decide whether to upgrade the public module version. The upgrade can be deferred. The DHCP service will continue to run with the original version of the public module until the administrator upgrades the module.

Data Service Configuration and DHCP Management Tools

The `dhcpmgr` and `dhcpconfig` management tools provide DHCP service configuration capabilities to system administrators. If you want your module to be available to users of the tools so they can configure the underlying data service, you must provide a JavaBeans™ component, known as a bean, for the public module.

The bean provides the public module with the context necessary to set the `PATH` variable, and optionally the `RESOURCE_CONFIG` variable, in `dhcpsvc.conf`.

Public Module Management Bean API Functions

The `dhcpmgr` tool provides an interface, `com/sun/dhcpmgr/client/DSModule`, which defines the API functions that the public module management bean must implement.

The `DSModule` interface is contained in the `dhcpmgr.jar` file. In order to compile the bean against this interface, you must add `/usr/sadm/admin/dhcpmgr/dhcpmgr.jar` to the `javac` class path. For example, for your bean named `myModule.java`, type

```
javac -classpath /usr/sadm/admin/dhcpmgr/dhcpmgr.jar myModule.java
```

getComponent()

Synopsis

```
abstract java.awt.Component getComponent()
```

Description

Returns a component that is displayed as one of the wizard steps for the DHCP Configuration Wizard. The returned component should be a panel containing GUI components to be used to obtain data-store-specific data from the user during configuration. The configuration data itself will be returned to the wizard as a result of calls to the `getPath()` and `getAdditional()` methods. See "getPath()" on page 26 and "getAdditional()" on page 26 for more information.

getDescription()

Synopsis

```
abstract java.lang.String getDescription()
```

Description

Returns a description that is used by the DHCP Configuration Wizard when it adds the data store to the list of data store selections. For example, the management bean for the ds_SUNWfiles.so public module returns "Text files" as the description.

getPath()

Synopsis

```
abstract java.lang.String getPath()
```

Description

Returns the path/location that is used by the data store (the PATH value in the Framework Configuration Layer configuration file /etc/inet/dhcpsvc.conf), or null if not set. The path/location value should be supplied by the user by interaction with the management bean's component. See "Passing Data Store Configuration Data" on page 24.

getAdditional()

Synopsis

```
abstract java.lang.String getAdditional()
```

Description

Returns additional data-store-specific information, such as the RESOURCE_CONFIG value in the Framework Configuration Layer configuration file /etc/inet/dhcpsvc.conf. The value returned by this method is most likely supplied by the user by interaction with the management bean's component. See "Passing Data Store Configuration Data" on page 24.

Public Module Management Bean Packaging Requirements

Public module management beans must meet the following packaging requirements.

- The public module management bean must be archived as a JAR file. The name of the JAR file must consist of the name of the public module and a .jar suffix. For example, the name of the public module management bean for the ds_SUNWfiles.so public module is SUNWfiles.jar.

- The JAR file must contain a manifest that identifies the bean class. For example, the manifest for the SUNWfiles.jar JAR file contains:

```
Name: com/sun/dhcpmgr/client/SUNWfiles/SUNWfiles.class
```

```
Java-Bean: True
```

The `com/sun/dhcpmgr/client/SUNWfiles/SUNWfiles.class` class is the Java class that implements the `com/sun/dhcpmgr/client/DSModule` interface.

3

Service Provider Layer API

This chapter lists and describes the API functions exported by public modules and consumed by the Framework Configuration Layer. The functions are grouped in sections according to their purpose. Within each section, functions are listed in an order in which you might use them.

The following topics are included:

- "General Data Store Functions" on page 29
- "dhcptab Functions" on page 32
- "DHCP Network Container Functions" on page 37
- "Generic Error Codes" on page 41

All implementations that match a certain Service Provider Layer API version must follow this specification for the API functions they implement. Later versions of the API must be backward-compatible with earlier versions. This means that additional API calls may be added, but existing ones cannot be changed or deleted.

See the include file /usr/include/dhcp_svc_public.h for more details about the functions.

General Data Store Functions

This section lists functions related to general data store activities.

`configure()`

Purpose

To pass a configuration string to the data store.

Synopsis

```
int configure(const char *configp);
```

Description

The configure() function is optional. If it is provided together with the required public module management bean (see "Data Service Configuration and DHCP Management Tools" on page 25), the Framework Configuration Layer calls this function when the public module loads, and passes in the public-module-specific configuration string, which is cached by the Framework Configuration Layer on the DHCP server for the data store module.

Returns

DSVC_SUCCESS, DSVC_MODULE_CFG_ERR

The configure() function returns DSVC_SUCCESS if the module wants the Framework Configuration Layer to continue to load the module, or DSVC_MODULE_CFG_ERR if the module wants the Framework Configuration Layer to fail the loading of the module. An example of such a situation is a configuration string so malformed that the required configuration of the module cannot take place.

mklocation()

Purpose

To create the directory where the data store containers are to reside.

Synopsis

```
int mklocation(const char *location);
```

Description

Creates the directory pointed to by location (if the directory does not exist) for data store containers to reside.

Returns

DSVC_SUCCESS, DSVC_ACCESS, DSVC_EXISTS, DSVC_BUSY, DSVC_INTERNAL, DSVC_UNSUPPORTED.

status()

Purpose

To obtain the general status of the data store.

Synopsis

```
int status(const char *location);
```

Description

The status() function instructs the data store to return its general status, and if location is non-NULL, further validates the location of the data store container by determining if the container does in fact exist, is accessible, and is formed correctly for the data store type. The data store must return the appropriate error codes if the facilities it needs are unavailable or it is otherwise not ready.

Returns

DSVC_SUCCESS, DSVC_ACCESS, DSVC_NO_LOCATION, DSVC_BUSY, DSVC_INTERNAL.

version()

Purpose

To obtain the version number of the API implemented by the data store.

Synopsis

```
int version(int *versionp);
```

Description

Data stores that support the Service Provider Layer API described in this manual are version 1 (one). The version is returned in the int pointed to by versionp.

Returns

DSVC_SUCCESS, DSVC_INTERNAL, DSVC_MODULE_ERR.

dhcptab **Functions**

The API functions described in this section are used with the dhcptab container.

list_dt()

Purpose

To list the name of the dhcptab container.

Synopsis

```
int list_dt(const char *location, char ***listppp, uint_t *count);
```

Description

Produces a dynamically allocated list of dhcptab container objects (listppp) found at location and stores the number of list items in count. If no dhcptab container objects exist, then DSVC_SUCCESS is returned, listppp is set to NULL, and count is set to 0.

Returns

DSVC_SUCCESS, DSVC_ACCESS, DSVC_NO_LOCATION.

open_dt()

Purpose

To open a dhcptab container or create a new one.

Synopsis

```
int open_dt(void **handpp, const char *location, uint_t flags);
```

Description

Opens an existing dhcptab container or creates a new container at location and initializes handp to point to the instance handle. Performs any initialization needed by the data store. When creating a new dhcptab, the caller's identity is used for owner/permissions. Valid flags include DSVC_CREATE, DSVC_READ, DSVC_WRITE, DSVC_NONBLOCK. Note that the creation of a dhcptab container as read-only (DSVC_CREATE | DSVC_READ) is invalid.

Returns

DSVC_SUCCESS, DSVC_EXISTS, DSVC_ACCESS, DSVC_NOENT, DSVC_NO_LOCATION,
DSVC_BUSY, DSVC_INTERNAL.

lookup_dt()

Purpose

To perform a lookup query for records in the dhcptab container.

Synopsis

```
int lookup_dt(void *handp, boolean_t partial, uint_t query, int count, const
dt_rec_t *targetp, dt_rec_list_t **resultp, uint_t *records);
```

Description

Searches the dhcptab container for instances that match the query described by the
combination of query and targetp. If the partial argument is B_TRUE, then partial query
results are acceptable to the caller. Thus, when partial is B_TRUE, any query that returns at least
one matching record is considered successful. When partial is B_FALSE, the query returns
DSVC_SUCCESS only if it has been applied to the entire container.

The query argument consists of 2 fields, each 16 bits long. The lower 16 bits select which fields
{key, type} of targetp are to be considered in the query. The upper 16 bits identify whether a
particular field value selected in the lower 16 bits must match (bit set) or not match (bit clear).
Bits 2 through 15 in both 16-bit fields are currently unused, and must be set to 0. Useful macros
for constructing queries can be found in Example 3–1.

The count field specifies the maximum number of matching records to return. A count value of
-1 requests the return of all records that match, regardless of the number. A count value of 0
causes lookup_dt to return immediately with no data.

resultp is set to point to the returned list of records. If resultp is NULL, then the caller is
simply interested in knowing how many records match the query. Note that these records are
dynamically allocated, and therefore the caller is responsible for freeing them. lookup_dt()
returns the number of matching records in the records argument. A records value of 0 means
that no records matched the query.

The following example includes macros you might find useful for constructing and
manipulating lookup queries for the DHCP network and dhcptab containers.

EXAMPLE 3–1   Useful Macros for Lookup Queries

```
/*
* Query macros - used for initializing query fields (lookup_d?)
*/
```

EXAMPLE 3–1   Useful Macros for Lookup Queries   *(Continued)*

```
/* dhcp network container */
#define DN_QCID 0x0001
#define DN_QCIP 0x0002
#define DN_QSIP 0x0004
#define DN_QLEASE 0x0008
#define DN_QMACRO 0x0010
#define DN_QFDYNAMIC 0x0020
#define DN_QFAUTOMATIC 0x0040
#define DN_QFMANUAL 0x0080
#define DN_QFUNUSABLE 0x0100
#define DN_QFBOOTP_ONLY 0x0200
#define DN_QALL (DN_QCID | DN_QCIP | DN_QSIP | DN_QLEASE | \
DN_QMACRO | DN_QFDYNAMIC DN_QFAUTOMATIC |\
DN_QFMANUAL | DN_QFUNUSABLE | \
DN_QFBOOTP_ONLY)

/* dhcptab */
#define DT_DHCPTAB "dhcptab"  /* default name of container */
#define DT_QKEY 0x01
#define DT_QTYPE 0x02
#define DT_QALL (DT_QKEY | DT_QTYPE)

/* general query macros */
#define DSVC_QINIT(q) ((q) = 0)
#define DSVC_QEQ(q, v) ((q) = ((q) | (v) | ((v) << 16)))
#define DSVC_QNEQ(q, v) ((q) = ((~(v << 16)) & (q)) | (v)))
#define DSVC_QISEQ(q, v) (((q) & (v)) && ((q) & ((v) << 16)))
#define DSVC_QISNEQ(q, v) (((q) & (v)) && (!((q) & ((v) << 16))))

/* Examples */
uint_t query;
/* search for dhcptab record with key value, but not flags value */
DSVC_QINIT(query);
DSVC_QEQ(query, DT_QKEY);
DSVC_QNEQ(query, DT_QTYPE);
/* search for dhcp network record that matches cid, client ip, server ip.
*/
DSVC_QINIT(query);
DSVC_QEQ(query, (DN_QCID | DN_QCIP | DN_QSIP));
```

Returns

DSVC_SUCCESS, DSVC_ACCESS, DSVC_BUSY, DSVC_INTERNAL.

add_dt()

Purpose

To add a record to the dhcptab container.

Synopsis

```
int add_dt(void *handp, dt_rec_t *newp);
```

Description

Adds the record newp to the dhcptab container referred to by handp. The signature associated
with newp is updated by the underlying public module. If an update collision occurs, the data
store is not updated. The caller is responsible for freeing any dynamically allocated arguments.

Returns

DSVC_SUCCESS, DSVC_ACCESS, DSVC_BUSY, DSVC_INTERNAL, DSVC_EXISTS.

modify_dt()

Purpose

To modify a record in the dhcptab container.

Synopsis

```
int modify_dt(void *handp, const dt_rec_t *origp, dt_rec_t *newp);
```

Description

Atomically modifies the record origp with the record newp in the dhcptab container referred to
by handp. The signature associated with newp is updated by the underlying public module. If an
update collision occurs, the data store is not updated. The caller is responsible for freeing any
dynamically allocated arguments.

Returns

DSVC_SUCCESS, DSVC_ACCESS, DSVC_BUSY, DSVC_COLLISION, DSVC_INTERNAL, DSVC_NOENT.

delete_dt()

Purpose

To delete a record from the dhcptab container.

Synopsis

```
int delete_dt(void *handp, const dt_rec_t *dtp);
```

Description

Deletes the record identified by the key, type and dt_sig fields of dtp from the dhcptab container referred to by the handle handp. If an update collision occurs, the matching record is not deleted from the data store, and DSVC_COLLISION is returned. The caller is responsible for freeing any dynamically allocated arguments.

If the dtp signature (dt_sig) is 0, the matching record is simply deleted with no detection of update collisions.

Returns

DSVC_SUCCESS, DSVC_ACCESS, DSVC_NOENT, DSVC_BUSY, DSVC_INTERNAL, DSVC_COLLISION.

close_dt()

Purpose

To close the dhcptab container.

Synopsis

```
int close_dt(void **handpp);
```

Description

Frees the instance handle and cleans up per-instance state.

Returns

DSVC_SUCCESS, DSVC_ACCESS, DSVC_INTERNAL.

remove_dt()

Purpose

To delete the dhcptab container from the data store location.

Synopsis

```
int remove_dt(const char *location);
```

Description

Removes the dhcptab container in location from the data store.

Returns

DSVC_SUCCESS, DSVC_ACCESS, DSVC_NOENT, DSVC_NO_LOCATION, DSVC_BUSY,
DSVC_INTERNAL.

DHCP Network Container Functions

The API functions described in this section are used to manipulate the DHCP network
containers and the IP address records within them.

list_dn()

Purpose

To return a list of network containers.

Synopsis

```
int list_dn(const char *location, char ***listppp, uint_t *count);
```

Description

Produces a dynamically allocated list of network container objects (listppp) found at
location and stores the number of list items in count. If no network container objects exist,
then DSVC_SUCCESS is returned, listppp is set to NULL, and count is set to 0.

Returns

DSVC_SUCCESS, DSVC_ACCESS, DSVC_NO_LOCATION.

open_dn()

Purpose

To open a network container or create a new one.

Synopsis

```
int open_dn(void **handpp, const char *location, uint_t flags, const struct
in_addr *netp, const struct in_addr *maskp);
```

Description

Opens an existing DHCP network container or creates a new container specified by netp and maskp (both host order) in location and initializes handpp to point to the instance handle. Performs any initialization needed by the data store. When creating a new DHCP network container, the caller's identity is used for owner/permissions. Valid flags include DSVC_CREATE, DSVC_READ, DSVC_WRITE, DSVC_NONBLOCK. Note that the creation of a DHCP network container as read-only (DSVC_CREATE | DSVC_READ) is invalid.

Returns

DSVC_SUCCESS, DSVC_EXISTS, DSVC_ACCESS, DSVC_NOENT, DSVC_NO_LOCATION, DSVC_BUSY, DSVC_INTERNAL, DSVC_UNSUPPORTED.

lookup_dn()

Purpose

To perform a lookup query for records in a DHCP network container.

Synopsis

```
int lookup_dn(void *handp, boolean_t partial, uint_t query, int count, const
dn_rec_t *targetp, dn_rec_list_t **resultp, uint_t *records);
```

Description

Searches a DHCP network container for instances that match the query described by the combination of query and targetp. If the partial argument is B_TRUE, then partial query results are acceptable to the caller. Thus, when partial is B_TRUE, any query that returns at least one matching record is considered successful. When partial is B_FALSE, the query returns DSVC_SUCCESS only if it has been applied to the entire container.

The query argument consists of 2 fields, each 16 bits long. The lower 16 bits select which fields {client id, flags, client IP, server IP, expiration, macro, or comment} of targetp are to be considered in the query. The upper 16 bits identify whether a particular field value selected in the lower 16 bits must match (bit set) or not match (bit clear). Bits 7 through 15 in both 16-bit fields are currently unused, and must be set to 0. Useful macros for constructing queries can be found in Example 3–1.

The count field specifies the maximum number of matching records to return. A count value of -1 requests the return of all records that match, regardless of the number. A count value of 0 causes lookup_dn to return immediately with no data.

resultp is set to point to the returned list of records. If resultp is NULL, then the caller is simply interested in knowing how many records match the query. Note that these records are dynamically allocated, and therefore the caller is responsible for freeing them. lookup_dn() returns the number of matching records in the records argument. A records value of 0 means that no records matched the query.

Returns

DSVC_SUCCESS, DSVC_ACCESS, DSVC_BUSY, DSVC_INTERNAL.

add_dn()

Purpose

To add a record to the DHCP network container.

Synopsis

```
int add_dn(void *handp, dn_rec_t *newp);
```

Description

Adds the record newp to the DHCP network container referred to by the handle handp. The signature associated with newp is updated by the underlying public module. If an update collision occurs, the data store is not updated.

Returns

DSVC_SUCCESS, DSVC_ACCESS, DSVC_BUSY, DSVC_INTERNAL, DSVC_EXISTS.

modify_dn()

Purpose

To modify a record in a DHCP network container.

Synopsis

```
int modify_dn(void *handp, const dn_rec_t *origp, dn_rec_t *newp);
```

Description

Atomically modifies the record origp with the record newp in the DHCP network container referred to by the handle handp. The signature associated with newp is updated by the underlying public module. If an update collision occurs, the data store is not updated.

Returns

`DSVC_SUCCESS, DSVC_ACCESS, DSVC_BUSY, DSVC_COLLISION, DSVC_INTERNAL, DSVC_NOENT.`

delete_dn()

Purpose

To delete a record from a DHCP network container.

Synopsis

`int delete_dn(void *handp, const dn_rec_t *pnp);`

Description

Deletes the record identified by the `dn_cip` and `dn_sig` elements of `pnp` from the DHCP network container referred to by the handle `handp`. If an update collision occurs, the matching record is not deleted from the data store and `DSVC_COLLISION` is returned.

If the `dn_sig` signature of `pnp` is 0, the matching record is simply deleted with no detection of update collisions.

Returns

`DSVC_SUCCESS, DSVC_ACCESS, DSVC_NOENT, DSVC_BUSY, DSVC_INTERNAL, DSVC_COLLISION.`

close_dn()

Purpose

To close the network container.

Synopsis

`int close_dn(void **handpp);`

Description

Frees the instance handle and cleans up per-instance state.

Returns

`DSVC_SUCCESS, DSVC_ACCESS, DSVC_INTERNAL.`

remove_dn()

Purpose

To delete the DHCP network container from the data store location.

Synopsis

```
int remove_dn(const char *location, const struct in_addr *netp);
```

Description

Removes DHCP network container netp (host order) in location.

Returns

DSVC_SUCCESS, DSVC_ACCESS, DSVC_NOENT, DSVC_NO_LOCATION, DSVC_BUSY,
DSVC_INTERNAL.

Generic Error Codes

The Framework Configuration Layer and Service Provider Layer API functions will return the following integer error values. Note that the file /usr/include/dhcp_svc_public.h is the definitive source for these codes.

```
 * Standard interface errors
*/
#define DSVC_SUCCESS          0  /* success */
#define DSVC_EXISTS           1  /* object already exists */
#define DSVC_ACCESS           2  /* access denied */
#define DSVC_NO_CRED          3  /* No underlying credential */
#define DSVC_NOENT            4  /* object doesn't exist */
#define DSVC_BUSY             5  /* object temporarily busy (again) */
#define DSVC_INVAL            6  /* invalid argument(s) */
#define DSVC_INTERNAL         7  /* internal data store error */
#define DSVC_UNAVAILABLE      8  /* underlying service required by */
                                 /* public module unavailable */
#define DSVC_COLLISION        9  /* update collision */
#define DSVC_UNSUPPORTED     10  /* operation not supported */
#define DSVC_NO_MEMORY       11  /* operation ran out of memory */
#define DSVC_NO_RESOURCES    12  /* non-memory resources unavailable */
#define DSVC_BAD_RESOURCE    13  /* malformed/missing RESOURCE setting */
#define DSVC_BAD_PATH        14  /* malformed/missing PATH setting */
#define DSVC_MODULE_VERSION  15  /* public module version mismatch */
#define DSVC_MODULE_ERR      16  /* internal public module error */
#define DSVC_MODULE_LOAD_ERR 17  /* error loading public module */
#define DSVC_MODULE_UNLOAD_ERR 18 /* error unloading public module */
#define DSVC_MODULE_CFG_ERR  19  /* module configuration failure */
#define DSVC_SYNCH_ERR       20  /* error in synchronization protocol */
```

```
#define DSVC_NO_LOCKMGR      21  /* cannot contact lock manager */
#define DSVC_NO_LOCATION     22  /* location nonexistent */
#define DSVC_BAD_CONVER      23  /* malformed/missing CONVER setting */
#define DSVC_NO_TABLE        24  /* table does not exist */
#define DSVC_TABLE_EXISTS    25  /* table already exists */
#define DSVC_NERR                (DSVC_TABLE_EXISTS + 1)
```

4

Code Samples and Testing

This chapter includes some segments of code that illustrate proper use of the API functions.

The following topics are included:

Code Templates

This section provides templates that show *in general* how you might use the API functions.

Note – Download the source code for Sun's ASCII files data store (ds_SUNWfiles) in the developer pages on Sun's web site (http://www.sun.com/developer). The source code for the module may prove invaluable in writing your own module.

General API Functions

This template uses the general API functions status(), version(), and mklocation().

EXAMPLE 4–1 general.c

```
* Copyright (c) 2000 by Sun Microsystems, Inc. /*
 * Copyright (c) 2000 by Sun Microsystems, Inc.
 * All rights reserved.
 */

#pragma    ident    "@(#)general.c    1.15    00/08/16 SMI"

/*
```

EXAMPLE 4–1 general.c *(Continued)*

```
 * This module contains the public APIs for status, version, and mklocation.
 */

#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <dhcp_svc_public.h>

/*
 * This API function instructs the underlying datastore to return its
 * general status. If the "location" argument is non-NULL, the function
 * validates the location for the data store containers (is it formed
 * correctly for the data store, and does it exist).
 */
int
status(const char *location)
{
    return (DSVC_UNSUPPORTED);
}

/*
 * Return the data store API version supported by this module.  This version
 * was implemented to support version 1 of the API.
 */
int
version(int *vp)
{
    *vp = DSVC_PUBLIC_VERSION;
    return (DSVC_SUCCESS);
}

/*
 * Create the datastore-specific "location" if it doesn't already exist.
 * Containers will ultimately be created there.
 */
int
mklocation(const char *location)
{
    return (DSVC_UNSUPPORTED);
}
```

dhcptab API Functions

This template illustrates functions that are used with the dhcptab container.

EXAMPLE 4–2 dhcptab.c

```
/*
 * Copyright (c) 1998-2000 by Sun Microsystems, Inc.
 * All rights reserved.
 */
```

EXAMPLE 4–2 dhcptab.c   *(Continued)*

```
#pragma    ident    "@(#)dhcptab.c    1.12    00/08/16 SMI"

/*
 * This module contains the public API functions for managing the dhcptab
 * container.
 */

#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <dhcp_svc_public.h>

/*
 * List the current number of dhcptab container objects located at
 * "location" in "listppp". Return number of list elements in "count".
 * If no objects exist, then "count" is set to 0 and DSVC_SUCCESS is
 * returned.
 *
 * This function will block waiting for a result, if the underlying
 * data store is busy.
 */
int
list_dt(const char *location, char ***listppp, uint32_t *count)
{
    return (DSVC_UNSUPPORTED);
}

/*
 * Creates or opens the dhcptab container in "location" and initializes
 * "handlep" to point to the instance handle. When creating a new dhcptab,
 * the caller's identity is used for owner/permissions. Performs any
 * initialization needed by data store.
 */
int
open_dt(void **handlep, const char *location, uint32_t flags)
{
    return (DSVC_UNSUPPORTED);
}

/*
 * Frees instance handle, cleans up per instance state.
 */
int
close_dt(void **handlep)
{
    return (DSVC_UNSUPPORTED);
}

/*
 * Remove dhcptab container in "location" from data store. If the underlying
 * data store is busy, this function will block.
 */
int
remove_dt(const char *location)
{
```

EXAMPLE 4–2 dhcptab.c *(Continued)*

```
    return (DSVC_UNSUPPORTED);
}

/*
 * Searches the dhcptab container for instances that match the query
 * described by the combination of query and targetp.  If the partial
 * argument is true, then lookup operations that are unable to
 * complete entirely are allowed (and considered successful).  The
 * query argument consists of 2 fields, each 16 bits long.  The lower
 * 16 bits selects which fields {key, flags} of targetp are to be
 * considered in the query.  The upper 16 bits identifies whether a
 * particular field value must match (bit set) or not match (bit
 * clear).  Bits 2-15 in both 16 bit fields are currently unused, and
 * must be set to 0.  The count field specifies the maximum number of
 * matching records to return, or -1 if any number of records may be
 * returned.  The recordsp argument is set to point to the resulting
 * list of records; if recordsp is passed in as NULL then no records
 * are actually returned. Note that these records are dynamically
 * allocated, thus the caller is responsible for freeing them.  The
 * number of records found is returned in nrecordsp; a value of 0
 * means that no records matched the query.
 */
int
lookup_dt(void *handle, boolean_t partial, uint32_t query, int32_t count,
    const dt_rec_t *targetp, dt_rec_list_t **recordsp, uint32_t *nrecordsp)
{
    return (DSVC_UNSUPPORTED);
}

/*
 * Add the record pointed to by "addp" to from the dhcptab container
 * referred to by the handle. The underlying public module will set
 * "addp's" signature as part of the data store operation.
 */
int
add_dt(void *handle, dt_rec_t *addp)
{
    return (DSVC_UNSUPPORTED);
}

/*
 * Atomically modify the record "origp" with the record "newp" in the
 * dhcptab container referred to by the handle.  "newp's" signature will
 * be set by the underlying public module.  If an update collision
 * occurs, either because "origp's" signature in the data store has changed
 * or "newp" would overwrite an existing record, DSVC_COLLISION is
 * returned and no update of the data store occurs.
 */
int
modify_dt(void *handle, const dt_rec_t *origp, dt_rec_t *newp)
{
    return (DSVC_UNSUPPORTED);
}

/*
 * Delete the record referred to by dtp from the dhcptab container
```

EXAMPLE 4–2 dhcptab.c   *(Continued)*

```
 * referred to by the handle. If "dtp's" signature is zero, the
 * caller is not interested in checking for collisions, and the record
 * should simply be deleted if it exists. If the signature is non-zero,
 * and the signature of the data store version of this record do not match,
 * an update collision occurs, no deletion of matching record in data store
 * is done, and DSVC_COLLISION is returned.
 */
int
delete_dt(void *handle, const dt_rec_t *dtp)
{
    return (DSVC_UNSUPPORTED);
}
```

DHCP Network Container API Functions

This template illustrates functions used with the the DHCP network containers.

EXAMPLE 4–3 dhcp_network.c

```
/*
 * Copyright (c) 1998-2000 by Sun Microsystems, Inc.
 * All rights reserved.
 */

#pragma    ident    "@(#)dhcp_network.c    1.12    00/08/16 SMI"

/*
 * This module contains public API functions for managing dhcp network
 * containers.
 */

#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <dhcp_svc_public.h>

/*
 * List the current number of dhcp network container objects located at
 * "location" in "listppp". Return number of list elements in "count".
 * If no objects exist, then "count" is set to 0 and DSVC_SUCCESS is
 * returned.
 *
 * This function will block if the underlying data service is busy or is
 * otherwise unvailable.
 */
int
list_dn(const char *location, char ***listppp, uint32_t *count)
{
    return (DSVC_UNSUPPORTED);
}

/*
```

EXAMPLE 4–3 dhcp_network.c   *(Continued)*

```
 * Creates or opens the dhcp network container "netp" (host order) in
 * "location" and initializes "handlep" to point to the instance handle.
 * Performs any initialization needed by data store. New containers are
 * created with the identity of the caller.
 */
int
open_dn(void **handlep, const char *location, uint32_t flags,
    const struct in_addr *netp)
{
    return (DSVC_UNSUPPORTED);
}

/*
 * Frees instance handle, cleans up per instance state.
 */
int
close_dn(void **handlep)
{
    return (DSVC_UNSUPPORTED);
}

/*
 * Remove DHCP network container "netp" (host order) in location.
 * This function will block if the underlying data service is busy or
 * otherwise unavailable.
 */
int
remove_dn(const char *location, const struct in_addr *netp)
{
    return (DSVC_UNSUPPORTED);
}

/*
 * Searches DHCP network container for instances that match the query
 * described by the combination of query and targetp.  If the partial
 * argument is true, then lookup operations that are unable to
 * complete entirely are allowed (and considered successful).  The
 * query argument consists of 2 fields, each 16 bits long.  The lower
 * 16 bits selects which fields {client_id, flags, client_ip,
 * server_ip, expiration, macro, or comment} of targetp are to be
 * considered in the query.  The upper 16 bits identifies whether a
 * particular field value must match (bit set) or not match (bit
 * clear).  Bits 7-15 in both 16 bit fields are currently unused, and
 * must be set to 0.  The count field specifies the maximum number of
 * matching records to return, or -1 if any number of records may be
 * returned.  The recordsp argument is set to point to the resulting
 * list of records; if recordsp is passed in as NULL then no records
 * are actually returned. Note that these records are dynamically
 * allocated, thus the caller is responsible for freeing them.  The
 * number of records found is returned in nrecordsp; a value of 0 means
 * that no records matched the query.
 */
int
lookup_dn(void *handle, boolean_t partial, uint32_t query, int32_t count,
    const dn_rec_t *targetp, dn_rec_list_t **recordsp, uint32_t *nrecordsp)
{
```

EXAMPLE 4–3 dhcp_network.c   *(Continued)*

```
    return (DSVC_UNSUPPORTED);
}

/*
 * Add the record pointed to by "addp" to from the dhcp network container
 * referred to by the handle.  The underlying public module will set
 * "addp's" signature as part of the data store operation.
 */
int
add_dn(void *handle, dn_rec_t *addp)
{
    return (DSVC_UNSUPPORTED);
}

/*
 * Atomically modify the record "origp" with the record "newp" in the dhcp
 * network container referred to by the handle.  "newp's" signature will
 * be set by the underlying public module.  If an update collision
 * occurs, either because "origp's" signature in the data store has changed
 * or "newp" would overwrite an preexisting record, DSVC_COLLISION is
 * returned and no update of the data store occurs.
 */
int
modify_dn(void *handle, const dn_rec_t *origp, dn_rec_t *newp)
{
    return (DSVC_UNSUPPORTED);
}

/*
 * Delete the record pointed to by "pnp" from the dhcp network container
 * referred to by the handle. If "pnp's" signature is zero, the caller
 * is not interested in checking for collisions, and the record should
 * simply be deleted if it exists. If the signature is non-zero, and the
 * signature of the data store version of this record do not match, an
 * update collision occurs, no deletion of any record is done, and
 * DSVC_COLLISION is returned.
 */
int
delete_dn(void *handle, const dn_rec_t *pnp)
{
    return (DSVC_UNSUPPORTED);
}
```

Testing the Public Module

See `http://www.sun.com/developer` for some downloadable test suites that may help you in testing your public module.

Index