# Vanity Naming Link ID and Persistent Configuration Management

Dan Groves

**Project Clearview I-Team**
clearview-discuss@opensolaris.org

Solaris Networking
Sun Microsystems, Inc.

Revision 1.15
July 11, 2007

# Contents

# 1 Introduction

As stated in [1], Vanity Naming will give Solaris administrators the ability to give a meaningful name to network links. Solaris components that need to work with these links will use an unique ID assigned to each link to access that link. The link ID makes accessing the link related information easier because the link ID always stays the same when the link name can change at any time.

`dlmgmtd(1M)` will manage the (link ID, link name) mapping and cache certain pieces of link configuration data. Persistent information about the (ID, name) mapping and with persistent link configuration will be stored in a persistent data store. Processes that need access to the data store will access it through one of the two APIs described later in this document. The APIs will access the data through `dlmgmtd(1M)`. The reason is to simplify locking and coordination between the user land and kernel land components that will need access to the information provided by this API. More details are in [1].

This document will describe how the (ID, name) mapping is persisted, the API used to access the (ID, name) mapping, the API used to access persistant link configuration information, and `dlmgmtd`.

# 2 Vanity Naming Storage

All persistent link data will be stored in a project private file called `/etc/dladm/datalink.conf`. Persistent link data is information about those links that must be preserved across reboots. Temporary link data will be stored in a project private file called `/var/run/datalink.conf`. Vanity Naming will introduce a new SMF service, `network/datalink-management`. This service will manage the `dlmgmtd` daemon. The daemon will manage the (ID, name) mapping and will be responsible for updating both copies of `datalink.conf`. `datalink.conf` will store the (ID, name) mapping and other link configuration data. Note that for temporary links the only data we expect to store in the temporary `datalink.conf` is the (ID, name) mapping. The only supported way for an end user to change contents of the repository is to use `dladm` from the command line. Others in the OS/Net consolidation may use the API described in this document to change the contents of the repository.

Persistent data and temporary data are stored in two separate files for ease of management. Temporary data must not be restored after a system reboot, but must be preserved in the event the `dlmgmtd` daemon is restarted for any reason. There is no easy when to tell if `dlmgmtd` has been started due to a system reboot or has been restarted because the administrator stopped the daemon. Files in `/var/run` are removed during a reboot, so the contents of `/var/run/datalink.conf` will be lost, as desired.

## 2.1 Dependencies

Both `network/physical` and `device/local` will depend on `network/datalink-management`. Therefore `dlmgmtd` is critical for proper booting and the `network/datalink-management` service will be part of the seed repository.

## 2.2 `datalink.conf`'s Format

Both `datalink.conf` files will have the same format. Both are plain text files, with one line of data per link. The format is:

```
link ID    prop0=type,val;prop1=type,val;...;propN=type,val;
```

An example `datalink.conf` is:

```
1    name=string,bge0;class=int,1;
2    name=string,bge1;class=int,1;
3    name=string,bge2;class=int,1;
4    name=string,vlan0;class=int,2;over=int,1;vlanid=int,123;
5    name=string,aggr0;class=int,4;nports=int,2;ports=string,2.3.;
```

The property name is a free-form string. Property values are stored as a (type, value) pair to facilitate converting data back from the string representation in the file to whatever format the API user passed the data into the API.

Note that the API will not perform any checks to ensure that the list of properties for a link make sense for a particular link. For example, the API will not prevent a user from storing properties that are specific to an aggregation for a link that is not an aggregation.

Entries are identified by the link ID in order renaming operations easier. This choice does make the file more confusing to an administrator who might view it because the given link name is buried in the property list. However, this is a private configuration file and editing the file directly is not supported.

As part of Vanity Naming, the existing `/etc/dladm/aggregation.conf` and `/etc/dladm/linkprop.conf` will be removed and converted to use the new repository.

## 2.3 Methods

As `dlmgmtd` is integral to using Vanity Naming, the start method will start `dlmgmtd`. During boot, the `network/datalink-management` start method will execute before any other networking related services start.

The stop method will kill `dlmgmtd`. Note that this is a dangerous operation, without `dlmgmtd` running the administrator cannot modify any link configuration data, the administrator cannot create links, physical devices cannot attach, IP interfaces cannot be plumbed via `ifconfig(1M)`, and there is nothing to respond to door upcalls related to link management from the kernel. However, link information stored in the kernel will not be affected, and when `dlmgmtd` restarts `dlmgmtd` will read in all link information from the repository. The refresh method will cause `dlmgmtd` to reload its data structures from both `datalink.conf` files.

# 3 API Externals

Access to the link data storage is through two consolidation private APIs. One API is used to manage the (ID, name) mapping, the other persists link configuration information (which includes the (ID, name) mapping). Changes made using the first API take affect immediately and are temporary (i.e. will not exist past the next reboot) unless persisted using the second API.

The API's purpose is to provide access to link configuration data while hiding details of how link configuration data is stored. The API's user does not, and should not, care how link configuration data is stored, and exposing this information could hamper efforts to improve on the implementation of the API. For example, even though the document talks about a particular method for persisting link configuration information, there is nothing exposed via the API that would require changes to API users if the method for persisting link configuration data changed.

Both APIs will be part of `libdladm`.

This section describes the part of the API that is exposed to other users. The next section goes into detail about the internal design of the API.

## 3.1 Types

### 3.1.1 `datalink_class_t`

This type identifies different link classes or groups of link classes supported. The type has the following values:

```
DATALINK_CLASS_PHYS
DATALINK_CLASS_VLAN
DATALINK_CLASS_AGGR
DATALINK_CLASS_ALL
```

Note that `DATALINK_CLASS_ALL` is used only for walking links (see the description of `dladm_walk_datalink_id` in the section on API calls).

This list includes only those link classes that exist at the time this document was written. The list may grow in the future.

### 3.1.2 `datalink_mediareq_t`

This type is used when walking links to determine which links the user is interested in. For example, the user can choose to walk only GLDv3 links, all links, just Ethernet links, or just WiFi links.

### 3.1.3 `datalink_id_t`

This type is used to store the unique link ID for a given link.

### 3.1.4 `dladm_conf_t`

This type represents a single link's configuration data. It is accessed via the functions listed below.

#### 3.1.5 dladm_datatype_t

This is an enumeration that describes the different types used by the persistent configuration API to store link configuration data. The possible values are:

```
DLADM_TYPE_STR
DLADM_TYPE_BOOLEAN
DLADM_TYPE_UINT64
```

### 3.2 Link ID Management API Calls

#### 3.2.1 dladm_create_datalink_id

```
dladm_status_t dladm_create_datalink_id(const char *name, datalink_class_t class, uint_t
media, uint32_t flags, datalink_id_t *id)
```

This function creates an (ID, name) mapping. The `flags` argument indicates whether the user intends to persist the mapping, and/or if the user wants the API to suggest a name. However, regardless of what `flags` is set to, the mapping is temporary until persisted by using the `dladm_*_conf` calls described later in this document. Possible values used to set `flags` are `DLADM_OPT_PERSIST`, `DLADM_OPT_ACTIVE`, and `DLADM_OPT_PREFIX`. If `DLADM_OPT_PERSIST` is set, then the API will assume that the user intends to persist the mapping, otherwise, the mapping is assumed to be temporary. `DLADM_OPT_ACTIVE` is set when the user intends to create a temporary link. `DLADM_OPT_PREFIX` indicates that the API should interpret `name` as a prefix and append a PPA to the prefix to create a unique name. In order to determine the new name, the user must call `dladm_datalink_id2info` on the returned ID.

The function returns `DLADM_STATUS_OK` on success, and an error code on error. Note that if `name` is NULL or zero length, if `class` is `DATALINK_CLASS_ALL`, if `media` is `DATALINK_MEDIA_ALL`, or if `id` is NULL, `DLADM_STATUS_BADARG` is returned. If a link with `name` already exists, the function will return an error.

#### 3.2.2 dladm_destroy_datalink_id

```
dladm_status_t dladm_destroy_datalink_id(datalink_id_t id, uint32_t flags)
```

This function destroys a (ID, name) mapping for the given link ID. The `flags` argument indicates whether the user intends to make the destruction persistent. If `flags` is `DLADM_OPT_PERSIST` the link ID is then available for reuse. However, the mapping will be restored after a reboot unless the API user calls `dladm_remove_conf`. If `flags` does not include `DLADM_OPT_PERSIST` and the mapping was created with `DLADM_OPT_PERSIST` then the link ID will not be available for reuse to prevent a possible ID collision. Passing `id` as `DLADM_INVALID_LINKID` will result in the function returning `DLADM_STATUS_BADARG`. The function returns `DLADM_STATUS_OK` on success.

#### 3.2.3 dladm_remap_datalink_id

```
dladm_status_t dladm_remap_datalink_id(datalink_id_t id, const char *name)
```

Change an (ID, name) mapping such that the given ID now maps to the given name. Changes are temporary unless persisted. If `id` is not a valid ID (either `DLADM_INVALID_LINKID` or an ID that was not returned from `dladm_create_datalink_id`) then the function returns `DLADM_STATUS_BADARG`.

If name is NULL or a zero length string, then DLADM_STATUS_BADARG is returned. The function will return DLADM_STATUS_OK on success.

### 3.2.4 dladm_name2info

```
dladm_status_t dladm_name2info(const char *name, datalink_id_t *id, uint32_t *flags,
datalink_class_t *class, uint32_t *media)
```

Retrieve the ID and class information for this name. If there is no link ID to name mapping for this name, an error is returned. If name is NULL or zero length, or id is NULL, then an error is returned. On success, the function returns DLADM_STATUS_SUCCESS, the ID corresponding to the given link name is returned in id, the link's class is returned in class, the link's media type is returned in media, and the flags used to create the link are returned in flags. Note that class, media, and flags can be NULL, in this case the NULL values are not returned.

### 3.2.5 dladm_datalink_id2info

```
dladm_status_t dladm_datalink_id2info(datalink_id_t id, uint32_t *flagp, datalink_class_t
*classp, uint_t *media, char *name, size_t name_length)
```

Retrieve the link information that corresponds to the given ID. If the given link ID has not been allocated, an error is returned. The name is returned in the caller supplied buffer name of size name_length, and is NULL terminated. The name is truncated if necessary to fit the name and the NULL terminator in the user supplied buffer. The link's class is returned in classp if classp is non-NULL. The link's media type is returned in media is media is non-NULL. The flags used to create the link are returned in flagp if flagp is non-NULL. The function returns DLADM_STATUS_BADARG if id is DLADM_INVALID_ID, if name is NULL, or if name_length is zero. The function returns DLADM_STATUS_OK on success.

### 3.2.6 dladm_walk_datalink_id

```
dladm_status_t dladm_walk_datalink_id(int (*fn)(datalink_id_t, void *), void *arg, datalink_class_t
class, datalink_mediareq_t media, uint32_t flags)
```

This function calls fn on links of the given class and media defined on the system. flags is used to determine if only persistent links are walked (flags is DLADM_OPT_PERSIST), only active links are walked (flags is DLADM_OPT_ACTIVE), or both types of links are walked (flags is DLADM_OPT_ACTIVE | DLADM_OPT_PERSIST). fn takes as an argument the value arg passed into dladm_walk_datalink_id and the link's name. fn must return either DLADM_WALK_TERMINATE or DLADM_WALK_CONTINUE which will indicate whether or not dladm_walk_datalink_id must continue walking the datalink IDs. The function returns DLADM_STATUS_OK on success, and a non-zero error code on failure. While passing arg as NULL is acceptable, passing fn as NULL will result in the function returning DLADM_STATUS_BADARG.

### 3.2.7 dladm_up_datalink_id

```
dladm_status_t dladm_up_datalink_id(datalink_id_t id)
```

Mark the given id as active. The function returns DLADM_STATUS_OK on success. If id is DLADM_INVALID_ID the function returns DLADM_STATUS_BADARG. If the given id does not exist, the function returns DLADM_STATUS_NOTFOUND.

## 3.3 Persistent Configuration Management API

All functions in this section affect the repository mentioned earlier. However, this does not allow for support of creating Jumpstart configurations or handling upgrades. The API provides no programmatic way to set configurations in an alternate root.

### 3.3.1 dladm_create_conf

```
dladm_status_t dladm_create_conf(const char *name, datalink_id_t id, datalink_class_t
class, uint32_t media, dladm_conf_t *conf)
```

Create a link configuration structure and initialize it with the given name, link ID, link class, and link media type. `flags` indicates if the information for the active or persistent (which might not be active) mapping is returned. The function will return the configuration structure in `conf` and return `DLADM_STATUS_OK` on success. If `name` is `NULL` or zero length, if `id` is `DLADM_INVALID_LINKID`, if `class` is `DATALINK_CLASS_ALL`, `media` is `DATALINK_MEDIA_ALL`, or if `conf` is `NULL`, the function will return `DLADM_STATUS_BADARG`. The configuration structure returned must be freed with `dladm_destroy_conf` to prevent resource leaks. The information stored in the configuration structure will not be persisted until `dladm_write_conf` is called.

### 3.3.2 dladm_read_conf

```
dladm_status_t dladm_read_conf(datalink_id_t id, dladm_conf_t *conf)
```

Retrieve the `dladm_conf_t` for the link with the given ID and store it in `conf`. The function returns `DLADM_STATUS_OK` on success and a non-zero error code on failure. If `conf` is NULL, then the function will return an error. The configuration structure returned must be freed with `dladm_destroy_conf` when the API user is done using it.

### 3.3.3 dladm_get_conf_field

```
dladm_status_t dladm_get_conf_field(dladm_conf_t conf, const char *field, dladm_datatype_t
*type, void *value, size_t vallen)
```

Get the data stored in field `field`. Data is returned in `value`, where `vallen` represents the length of the buffer. The data's type is returned in `type`, assuming `type` is not `NULL`. If `value` is not big enough to hold the requested data, an error is returned. If `value` is not properly aligned for the data requested, the function will not fault (however, the API user's code might fault) or return an error. Data returned as a string will be `NULL`-terminated. If the operation completed successfully, the function returns `DLADM_STATUS_OK`. Otherwise, it returns a non-zero error code. The following checks are performed on the input:

- `field` must be non-NULL and not a zero length string
- `value` must be non-NULL
- `vallen` large enough to hold the requested data

If any of these checks fail, the function returns `DLADM_STATUS_BADARG`. Note that it is not an error to pass `type` as `NULL`.

### 3.3.4 dladm_set_conf_field

```
dladm_status_t dladm_set_conf_field(dladm_conf_t conf, const char *field, dladm_data_t
type, void *value)
```

Update the properties structure for this link with the given value. `value` is the data the user wishes to add to the structure and `type` is its type. String data must be `NULL` terminated. The change is not persisted until `dladm_write_conf` is called on `conf`, and will only apply to this instance of `conf`. For example, if another call to `dladm_read_conf` is made on the same link, the new `conf` structure will not show any changes made to the original conf structure. If the operation succeeds, `DLADM_STATUS_OK` is returned. Otherwise, an error code is returned. If `value` is NULL, the function will fail. If `field` is NULL or a zero length string, the function will fail. It is an error to change the link ID, the name, or the link class.

### 3.3.5 dladm_unset_conf_field

```
dladm_status_t dladm_unset_conf_field(dladm_conf_t conf, const char *field)
```

Remove the given field from the given link configuration. The change does not take affect until persisted via a call to `dladm_write_conf`. The function returns `DLADM_STATUS_OK` on success. If `field` is `NULL` or zero length, the function returns `DLADM_STATUS_BADARG`. If the field does not exist in this configuration structure, the function returns `DLADM_STATUS_NOTFOUND`.

### 3.3.6 dladm_rename_conf

```
dladm_status_t dladm_rename_conf(datalink_id_t id, const char *name)
```

For the given ID, update its configuration structure to use the given name. This function will update persistent storage. If `id` is `DLADM_INVALID_LINKID`, if `name` is `NULL`, or if `name` is zero length, then the function will return `DLADM_STATUS_BADARG`. The function will return `DLADM_STATUS_OK` on success.

### 3.3.7 dladm_write_conf

```
dladm_status_t dladm_write_conf(dladm_conf_t conf)
```

Commit the datalink configuration pointed to by `conf` to storage. The function returns `DLADM_STATUS_OK` on success. If an update collision occured when trying to commit the configuration, then the function will return `DLADM_STATUS_TRYAGAIN` and the user will have to re-read the configuration structure and redo the changes to the structure. If the user attempts to write a configuration that would create a (link ID, name) mapping collision, then the function will return `DLADM_STATUS_EXIST`.

### 3.3.8 dladm_remove_conf

```
void dladm_remove_conf(datalink_id_t id)
```

Remove the configuration data for the link with the given ID from storage.

#### 3.3.9 dladm_destroy_conf

`void dladm_destroy_conf(dladm_conf_t conf)`

Release the `conf` structure allocated by `dladm_create_conf` or `dladm_read_conf`. This function does not affect anything in the persistent storage. The caller must not use `conf` after calling this function as `conf` is no longer valid.

## 3.4 Examples of API Usage

Consider the case where the API user wants to create a new interface that is persistent across reboots. The caller would write code similar to the following psuedocode:

```
create("net1")  /* Let's assume this is on physical link bge0 */
    dladm_create_datalink_id("net1", DATALINK_CLASS_PHYS,
        DATALINK_MEDIA_ETHER, DLADM_OPT_PERSIST | DLADM_OPT_ACTIVE, &id);
retry:
    dladm_create_conf("net1", id, DATALINK_CLASS_PHYS,
        DATALINK_MEDIA_ETHER, &link_conf);
    dladm_set_conf_field(link_conf, "device", DLADM_TYPE_STR,
        (void *)"bge0");
    rc = dladm_write_conf(link_conf);
    dladm_destroy_conf(link_conf);
    if (rc == DLADM_STATUS_TRYAGAIN)
        goto retry;
```

Renaming that interface would look like:

```
rename("net1", "net2")
    dladm_name2info("net1", &id, NULL, NULL);
    dladm_remap_datalink_id(id, "net2");
    dladm_rename_conf(id, "net2");
```

Deleting the newly renamed net2 interface:

```
remove("net2")
    dladm_name2info("net2", &id, NULL, NULL);
    dladm_destroy_datalink_id(id, DLADM_OPT_PERSIST);
    dladm_remove_conf(id);
```

Creating a temporary link looks like:

```
create_temp("temp0")
    dladm_create_datalink_id("temp0", DATALINK_CLASS_PHYS,
        DATALINK_CLASS_ETHER, 0, &id);
```

And then the user can delete the temporary link by doing the following:

```
delete_temp("temp0")
    dladm_name2info("temp0", &id, NULL, NULL);
    dladm_destroy_datalink_id(id, 0);
```

The following pseudocode gives an example of how to retrieve a link name and a link class from a configuration structure given a link ID:

```
get_name_and_class(id)
    dladm_read_conf(id, &link_conf);
    dladm_get_conf_field(link_conf, "name", &type, (void *)name,
        sizeof(name));
    dladm_get_conf_field(link_conf, "class", &type, (void *)&class,
        sizeof(class));
```

# References

[1] Vanity Naming and Nemo Unification High Level Design Specification
http://www.opensolaris.org/os/project/clearview/uv-design.pdf