Brussels - NIC configuration Design Specification
Revision : 1.2

Sowmini Varadhan, Raymond Li, Artem Kachitchkine
{sowmini.varadhan, raymond.li, artem.katcitchkine}@sun.com

October 4, 2007

Contents

1 Introduction

Currently, there is a lack of a uniform interface for configuring commonly used parameters for nic drivers. Some or all of the methods below are used to customize, or obtain status about, driver configuration:

- `driver.conf` (4)
- `ndd` (1M)
- via obp on sparc,
- `kstat` (1M) can be used to read existing configuration for some parameters

Moreover the same feature may sometimes be set via multiple means (e.g., both by `driver.conf` and `ndd`).

All of the diverse methods for tuning parameters makes system/driver configuration a confusing and arcane process that has been known to frustrate both our customers and even engineers attempting to use driver features. The lack of a stable, standard NIC driver interface also makes it difficult/impossible to automate testing of NIC features.

Moreover, each of the methods above has limitations. Configuration settings via `ndd` (1M) are not persistent across reboots, and `ndd` (1M) itself is an undocumented programming interface with several constraints: the input to the "set" command may be scalar only, and the output from the "get" command is limited to 64 KB (see CR 4326707). The syntax for `driver.conf` (4) is not standardized or documented across drivers so that one has to search through driver sources to find the syntax for a given parameter. For example, the various parameters that are used across a sampling of driver to enable Jumbo Frame support

driver	parameter
e1000g	MaxFrameSize
xge	default_mtu
ixgb	default_mtu
bge	default_mtu, but may be impacted by kernel global bge_jumbo_enable
ce	accept_jumbo

Project "Nemo", aka GLDv3[4] provides a path for a more unified solution to this problem: it would be cleaner to provide a uniform interface to configuring drivers by using the `dld` layer as the intermediary. This solution would be consistent with the direction indicated by [8] and other recent features such as IEEE 802.3ad support where `dladm` is the administrative tool chosen.

The aim of the Brussels project is to establish a uniform administrative interface across GLDv3 drivers that is conformant with the Nemo framework.

1.1 Competitive Analysis

Both BSD and Linux use the `sysctl()` command for tuning kernel parameters. The `sysctl()` command is similar to `ndd` , but is capable of returning information about non-scalar parameters like strings and tables. Persistence across reboots is supported via `/etc/sysctl.conf`.

In addition, Linux also supports the `ethtool(8)` command [1] to display or change ethernet card settings for a well-defined set of properties. Many of the commonly tweaked BSD ethernet configuration parameters are set via `ifconfig`.

On HPUX, the System Administration Manager [3] program provides a menu-driven GUI for configuring network/device parameters.

AIX provides the "`no`" (network options) command to tune Layer 2 and Layer 3 parameters. Driver parameters are treated as "attributes" and the `lsattr`, `chattr chdev` utilities for getting/setting driver configuration parameters [2]. These commands allow the user to get/set default and effective information, as well as get the range of allowed values. In general, AIX offers a systematic and user-friendly method for managing system tunables: every subsystem *foo* is associated with intuitive commands like `ch`*foo*, `ls`*foo*, `rm`*foo*. The System Management Interface tool (SMIT) on AIX offers a menu-driven interface to these tools that may be run both in ASCII and in graphical mode.

1.2 What we want to preserve

Although each of the existing methods has clear drawbacks, these methods also have some desirable properties that must be retained in a new proposal.

- `ndd` is perceived by customers as a quick way to set config variables on a specific instance of a driver on the fly. It is frequently used to set engineering interfaces for tweaking obscure parameters in the underlying implementation.
- `driver.conf` tunables persist across reboot, and are configured early in the driver attach process.
- `'ndd -get ... \?'` is an extremely popular way of finding the current list of available parameters.
- `ndd` has built-in range checks, which quickly identify input that is out of range.

2 Proposal

Project "Nemo", aka GLDv3 [4], is the current cutting-edge network device driver framework. In addition to providing a unified model for device driver, Nemo introduced the tool, `/sbin/dladm`, to administer GLDv3 links and features by issuing system calls to the kernel's `dld` module.

The first phase of this project will involve the provision of a generic configuration management interface for GLDv3[1] drivers via `dladm`. This work would comprise two major pieces: standardizing driver configuration via `dladm`, and providing the needed capability from network drivers controlled by Sun.

Subsequent phases of the project would provide similar cleanup with respect to kstat and ioctl handling in the Nemo and driver layers, and also examine the usage of `ndd` (1M) and `system`(4) in the Network and Transport layer, with the objective of sorting out the various tunables, and providing alternate solutions where appropriate. Configuration mechanisms such as BSD's `sysctl`(8) should also be evaluated to investigate the applicability to Solaris.

2.1 Definitions

The following definitions are used in the remainder of this document.

- A **"property"** is any network driver tunable such as jumbo frame support, link speed etc. Properties will only be associated with data-links which correspond to "MAC-registered" entities, i.e., entities that register as a unique MAC.
- **"Well-known"**, or PUBLIC properties that characterize behavior of drivers of the given media type. Examples of such properties are link speed, auto-negotiation for ethernet.
- **"Driver-specific"**, or PRIVATE, properties that are peculiar to a certain class of drivers. These properties may be driver specific because they are closely related to the hardware associated with the driver (e.g, WiFi properties) or may be (potentially debugging related) properties that are related to the implementation itself (e.g., "infinite burst" in ce).

Properties tuned for Network drivers are characterized by the potential involvement of the Administrator in setting their values. The value that would be appropriate for the property is typically impacted by several factors including local hardware capabilities and the capabilities advertised by the peer on the network. Unlike hardware capabilities that are negotiated via the `mac_capab_get()` interfaces (see, for example, PSARC 2006/265), auto-tuning of properties may frequently not be possible for valid network configurations, and the property may need to be manually set to non-default values based on administrative preferences. The distinguishing difference between properties and capabilities is that the latter may never be set, but the former may require setting in many configurations.

A side-effect of the association of properties with data-links is that this proposal will not focus on methods to set properties on a per-driver basis, though the design proposed here may be extended to set driver-wide parameters.

2.2 Requirements

In addition to preserving the desirable features listed in Section 1.2, the new proposal must meet the following list of requirements.

1. Define a standardized interface via GLDv3 to configure tunable properties for network drivers. The new interface should preserve all the desirable properties in existing methods (See section 2.2) while building upon, and generalizing, the existing features in the Nemo framework (e.g., those introduced by [7]).
2. Provide support for well-known properties (e.g., enable/disable jumbo frame support) in the GLDv3 framework.

[1] non-Nemo drivers will be supported via softmac developed as part of Project Clearview[5]

3. Modify internal NIC drivers as needed for these well-known properties. The project will modify all internal GLDv3 drivers, starting with the most commonly used ones: e1000g, nxge, nge, ixgb, xge, rge, bge;

4. Provide extensible support for driver-specific properties, along with associated documentation, to support general driver configuration parameters.

5. Enable association of interface stability levels with driver configuration properties (See `attributes`(5)).

6. Provide methods to obtain a summary of currently configured properties that includes the name of the property, a short description of the property, and values.

7. Provide `mdb` macros to extract a summary of the configuration.

8. Publish a "Best Practices" document that documents the requirements for supporting well-known properties in third-party drivers.

9. Provide a GUI for driver configuration, that includes support for both well-known and private properties.

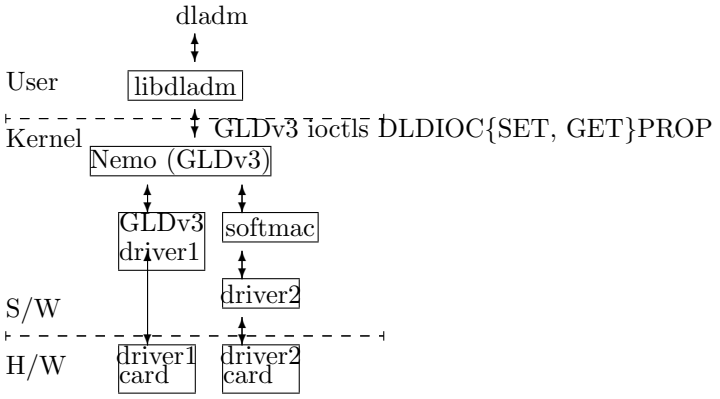

Figure 1: Flow diagram for device property administration; "driver1" is an instance of a GLDv3 driver, "driver2" is an instance of a GLDv2 driver

3 Unification of device driver framework

We propose to unify nic administration at the kernel Nemo layer for MAC properties through the following modifications.

1. introduce callback functions that every GLDv3 driver provides when registering via `mac_register()`. These callback functions will be invoked when the dld layer wishes to set/reset/get a property. This is further discussed in Section 3.1

2. modify dladm/libdladm suitably so that queries to set/reset/get props from user-space occur as illustrated in Figure 1. This is further discussed in Section 3.2

3.1 Driver changes

Currently, every GLDv3 driver registers itself by invoking `mac_register()` with a pointer to a `mac_register_t` as shown in Figure 2. The Brussels proposal will add two new fields

```
typedef int (*mac_propf_t)(void *mi_driver,              uint_t pr_num,
     int proplen, void *propval, /* ... */);
mac_spropf_t mc_setprop;
mac_gpropf_t mc_getprop;
```

Drivers that do not wish to support Brussels properties may provide null values for `mc_setprop` and `mc_getprop` during mac registration. Corresponding to the new fields, two new flags, `MC_SETPROP` and `MC_GETPROP`, will be added to indicate (via `mc_callbacks`) that the appropriate member function is valid. When it is valid, the member function is invoked with first argument pointing at the `mi_driver` member of the `mac_impl_t` for the interface, and the remaining arguments derived from the `dld_ioc_prop_val_t` structure described in Section 3.2.1.

3.2 Extensions to dladm and system calls

Extensions for WiFi configuration via `dladm` (1M) were introduced in [7]. Three subcommands for link administration were introduces; implementation details of these commands are available in [9]. Building upon this model, the {set, reset, show}-linkprop commands will be extended as described in [8] for "MAC-registered" network drivers.

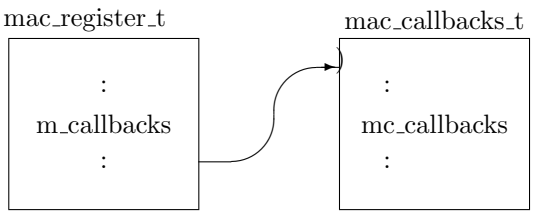

Figure 2: GLDv3 structures provided to mac_register()

Two new ioctls, DLDIOCSETPROP/DLDIOCGETPROP will be introduced and libdladm will be modified to send these ioctls on the `DLD_CONTROL_DEV`. These ioctls will be processed in the kernel `dld` [2] and `(*mc_setprop)()` or `(*mc_getprop)()` will be invoked as described in Section 3.1.

3.2.1 Data passed in {set, get}prop ioctl

The libdladm library manages a set of attributes associated with each property. These attributes include the following.

1. `dld_pr_num`, an encoded scalar representation of the property (e.g., `DLD_PROP_ADV_AUTONEG`), where the valid set of scalar encodings is available in `<sys/dld.h>`,
2. `dld_pr_name`, string representation of the property name (e.g., "`adv_autoneg_cap`"),
3. a string containing a short description of the property (e.g., "`Advertise auto-negotiation`"),
4. `(*pd_set)()/(*pd_get)()`, property-specific member function pointers for constructing and parsing the data to be sent for set/get ioctls (see PSARC 2007/140),
5. `(*pd_check)()` property-specific member function pointer for doing range/value checks of input data to be sent to the kernel (see PSARC 2007/140).

The data sent from user-space to kernel thus has the following format:

```
typedef struct dld_ioc_prop_val_t {
     int     pr_version; /* <sys/dld.h>
     char    pr_linkname[LIFNAMSIZ];
     uint_t  pr_num;                 /* obtained from <sys/dld.h> */
     uint_t  pr_name[DLD_LINKPROP_NAME_MAX];    /* for Private properties */
     uint_t  pr_valsize;             /* sizeof pr_val */
     void    *pr_val;
} dld_ioc_prop_val_t;
```

The property is identified by the `pr_num` for Public properties, and by the `pr_name` for Private properties. Note that the type of the data associated with the property is implicitly known to the member functions in the attributes above.

[2] in drv_ioc()

The `pr_version` field is intended to allow the consumer (the driver's `mc_setprop` or `mc_getprop`) to identify the format of the structure passed in. Valid constant values for this field are defined in `<sys/dld.h>`. If the driver does not recognize the `dld_pr_version` value passed to it, it should return EINVAL for that DLDIOCSETPROP/DLDIOCGETPROP ioctl.

The mapping from a string representation of property name to the property number will be done in libdladm for Public properties. In the case of Private properties, `/sbin/dladm` will pass down a value of "`DLD_PROP_PRIVATE`" for `pr_num`, and the string representation of the property name. The driver can then parse these values and interpret the input appropriately. Note that an addtional optimization to this scheme is possible if the driver registers all known properties with the framework when it starts up. Details of this optimization, the contents of `pr_val[size]` and the user-space processing of the ioctl are further described in Sections 3.2.3 and 3.2.4.

In the aftermath of the libdladm restructuring achieved by PSARC 2007/140, a command-line directive to set/get a property p on link l will cause properties to be searched in the following order:

1. search for p in the list of well-known, Public properties,
2. if not found, search for p in the `wlan` (wifi) properties,
3. if not found, pass down p as a Private property.

The reason that `wlan` properties are treated separately is that system-calls in this case are written as discrete ioctls to the net80211 module. The design issues in this case are historical constraints – specifically, wificonfig issues the same ioctls, and a desire to retain binary compatibility for code on opensolaris.org. As a result, some `wlan` properties will not be merged into the Brussels framework.

Conflicts between WiFi, Private and Public property names will be resolved by using the implicit preference from the ordering scheme above. Conversion of WiFi drivers like ath and wpi will provide `mc_setprop`/`mc_getprop` entry points for WiFi properties so that the search order will allow the eradication of the net80211 paths when compatibility issues are resolved.

3.2.2 Attributes(5) considerations

In the initial phase of Brussels, all properties will be delivered with "Volatile", or, at best, "Uncommitted" attributes with these attributes being modified as appropriate after additional user-experience has been acquired.

3.2.3 Public properties

The current list of Public properties is listed in Appendix C.

When setting Public properties, the size of the input data is related to the semantics of the property, so that the `libdladm` is able to validate the input value and send down typed data (e.g., mapping the string "half" to the integer value 1) to the driver.

If an attempt is made to set a Public property on a device that cannot support that property (e.g., attempting to set speed to 10M on the `xge` interface) the ENOTSUP error will be returned by the libdladm API, allowing dladm to then print a message to indicate that the property is not applicable for that driver (e.g., "–" in show-linkprop output, and a "Property not supported by driver" message for set-linkprop output) . Note that this is distinct from the EINVAL ("Invalid argument") error to be returned when the input value itself is malformed.

When the underlying driver has not plugged into the Brussels framework (i.e., the callback parameters, `mc_setprop` and `mc_getprop`, are null), the ENOTSUP error is returned on an attempt to do 'dladm show-linkprop'.

Public properties are logically subdivided into categories based on the link type, and the execution of the "`dladm show-linkprop <link>`" command for a link of a given link type will only display output that is relevant to that link type. Thus executing the `show-linkprop` command on a tunnel interface will not display any lines of output for MII properties. Section 7 discusses some implementation details for achieving the logical subdivision of Public properties based on link type.

3.2.4 Private Properties

Link properties that begin with "_" (the underscore) character are treated as Private Properties for the link, and subject to change or removal;

Private property support is intended to accomodate off-beat tunables that have value to a very limited audience for a minority of drivers. The support for private properties in the Brussels framework is provided with the expectation that, in general, driver-writers as well as Administrators desire common interfaces for all drivers, as opposed to arcane customizations for a specific driver. Appendix B provides an example implementation of a tunable property in the `bge` driver to accomodate deficiencies in the driver hardware.

In the initial implementation of the prototype, it is expected that the following properties will be available as Private properties. Note that these properties have been selected because they are currently heavily used for tuning/debugging customer configurations, but it is expected that future modifications to the IP/DLD layers will make them auto-tunable variables. Thus the private property implementation of these features is offered as a short-term solution that are expected to be removed in the future.

- interrupt coalescing parameters used in GLD's blanking mechanism. Note that it is expected that the Crossbow Project will obviate the need for these parameters in the near future.
- bcopy/dma threshold, that controls the threshold at which the driver switches rx/tx mode between bcopy and DMA (DVMA on sparc) binding to provide better performance when dealing with variable sized packet flows.

In the current prototype, the arguments provided to `/sbin/dladm set-linkprop` are sent down to dld as a string, and thence passed to `(*mc_setprop)()`. The driver may use functions such as `ddi_strtoul`(9f) to parse the input and perform the necessary actions.

Note that dladm can currently handle input of the form

```
#/sbin/dladm set-linkprop -p <prop>=<val1>,<val2>,<val3>,...,<valn>
```

where property `<prop>` is assigned an array of values `<val1>,...<valn>`. Input of this form will be passed down to the driver as a null-terminated string of comma-separated values for Private Properties.

An enhancement to this method is possible if the driver registers all known properties with the Brussels framework, so that the framework can then map strings to formatted input in every case. Thus, drivers that wish to leverage from the additional string-processing, and input validation facilities provided by the framework, and are able to register all properties during autoconfiguration may benefit from this enhancement.

3.2.5 Show Configuration

PSARC 2006/499 [**?**] has proposed several changes to the `dladm` subcommands. Brussels will extend that proposal to provide the following:

1. The `dladm show-linkprop` command will complement the `dladm set-linkprop` command for the r/w properties. Thus, if `set-linkprop` supports an invocation of the form

   ```
   # dladm set-linkprop -p foo_prop=<value> <link>
   ```

   then the output of `show-linkprop` would contain

   ```
   PROPERTY   VALUE     DEFAULT   POSSIBLE
   :
   foo_prop   <value>           ...
   :
   ```

```
# dladm show-ether bge1

LINK   PTYPE    STATE   AUTO   SPEED-DUPLEX   PAUSE
bge0   current  up      yes    1G-f           1
```

```
# dladm show-ether -x bge1

LINK   PTYPE    STATE   AUTO   SPEED-DUPLEX           PAUSE
bge0   current  up      yes    1G-f                   1
       capable  --      yes    1G-fh,100M-fh,10M-fh   1
       adv      --      yes    1G-fh                  1
       peeradv  --      yes    1G-fh                  1
```

Table 1: Sample output of `dladm show-ether` command. Units of speed are encoded in the speed-duplex result as G (Gigabit/s) or M (Mb/s). Duplex values are encoded as f (full-duplex) and h (half-duplex).

2. A new subcommand, `show-ether`, will be added to print a summary of the link state. Table3.2.5 provides tentative sample output proposed for the `show-ether` sub-command.

The interaction between `show-ether` and the "`-o all`" and "`-x`" flags are as follows.

- "`dladm show-ether`" will report the currently negotiated values of auto-negotiation, speed, duplex and pause. The existing definiton and semantics, e.g., pause/asmpause and rem_fault definitions, of some ethernet properties are complex and could use simplification. Alternative solutions, including the provision of cleaner semantics for these, are actively being investigated at the time of this writing.
- "`dladm show-ether -x`" will report three rows of output for each link, corresponding to the current, advertised, and link-partner advertised (peeradv) values.

4 Persistence or property settings

This section gives an overview of the "persistent property management" component of the Umbrella case described in PSARC 2007/429.

Solaris Nevada provides for persistent property configuration of WiFi and aggregated links by default via `dladm/libdladm`.

Unless the "`-t`" option has been specified, `dladm` automatically saves configuration information in `/etc/dladm/linkprop.conf`[3] so that, after the execution of a command of the form:

```
# dladm set-linkprop -p <prop_name>=<prop_value> <link>
```

results in the addition of an entry to `/etc/dladm/linkprop.conf`. The properties saved to the database are read after plumb in
`/lib/svc/method/net-physical` and the command sequence below is effectively used to add the saved properties.

```
# ifconfig <link> plumb
# /sbin/dladm init-linkprop
```

[3]Note that the UV segment of Project Clearview is making modifications to this feature. At the time of this writing, the proposal is to use an smf repository for managing persistent information.

The `net-physical` script waits until the interfaces are plumbed to restore link properties mainly due to unfortunate implementation details of WiFi drivers which reset all link properties during plumb. For Ethernet drivers, as long as we ensure that the `mc_start` routine does not reset any of the configured properties, the actual order of invocation of the "plumb" and "init-linkprop" pair is inconsequential.

The larger issue, currently unresolved for WiFi drivers, is to ensure persistence of link properties across a module unload/reload sequence (CR 6485961). Possible solutions to this issue are discussed in this Section.

4.1 Constraints on dynamic property restoration/modification

Currently, `driver.conf` properties are typically recovered and parsed during `attach(9E)`, and drivers rely on this setting to initialize state, e.g., drivers typically initialize several chip related properties, including those pertaining to memory allocation for Tx/Rx rings based on property settings read at the start of attach(9E). Although it would be ideal for the driver to be able to intercept messages that modify property values and reset any data-structures or state based on the new configuration for any property, the reality is that it may not be possible because the driver is being used, and has to defer the reset operation. In addition, when there are active DLS clients using a link, it may not always be possible to modify property values, if the clients are not equipped to reset their state based on the property change. In the case of modification to Tx/Rx rings on a plumbed interface, changes would require flow/squeue reassignment in IP.

A related case-study for modifications to Jumbo Frames is discussed in Appendix A.

Since the constraints placed on dynamic property modification are dependant on the implementation details of the driver, when an attempt is to modify a property that is syntactically correct, but cannot effected immediately, the driver's setprop function should return the EBUSY error code. The driver may additionally log a message informing the administrator about what is required to make the change effective. The libdladm module can then convert the EBUSY error code to an internal error code, and react appropriately so that the change will be effective on the next restart of the driver.

4.2 Property restoration during attach(9E)

To retain current behavior, interfaces will be provided in mac/dld layers to restore properties saved in linkprop.conf in the kernel. Drivers may then invoke these interfaces to restore settings early in their attach(9E) routine.

Restoration of properties via GLDv3 must preferably be done without retaining property information in the kernel for links that do not exist.

The solution proposed by Brussels is to have a listening daemon that watches for, and obtains notification on, attach events. The daemon would have to trigger the `init-linkprop` sequence when the link comes up. Since this method involves message passing from the kernel to user-space, followed by a series of `setprop` calls from user-space to kernel, the driver should get the property settings "at the right time", i.e., before the property value is used.

The following synchronous solution is proposed here:

1. Drivers will send an "I'm awake" message from the `xxattach` function to a user-space daemon `dladmd` via a `door_ki_upcall`[4], and block in the drivers `xxattach` routine until the init-linkprop is completed.

2. The receipt of the door call will cause `dladmd` to execute calls to libdladm that will result in the properties being loaded into the kernel in a list of `mac_prop_t` structures.

3. In order to access the value of Brussels properties, drivers will be modified to do a sequence of commands such as the example below, which demonstrates the retrieval of a `uint64` property:

```
err = mac_prop_init("driver", instance, &handle);
val = mac_prop_get_uint64("driver", "property");  val = mac_prop_fini(handle);
```

4. Properties associated with a driver instance will be unloaded when the last handle is released via `mac_prop_fini()`.

[4]Note that there are ongoing discussions to come up with a common daemon for Brussels, Clearview-UV and NWAM projects, all of which have similar needs for a daemon of this nature

4.3 Diskless boot considerations

Diskless boot environments add additional issues when the driver code has been inflexibly written to require the property value to be available early in attach: here, the attach may take place well before init, so that the presence of, and communication with, the `dladmd` daemon may not be relied upon for property restoration.

The situation closely parallels the analogous problem for Vanity Naming described in [**?**] and analogous solutions will be provided. Wherever possible, properties converted to Brussels will be implemented in the driver so that they may be modified by stopping and starting the driver, so that, even if the driver has completed the attach, the initialization scripts that start the driver can ensure that `dladm` configurations settings are loaded correctly by first resetting the driver.

In addition, as a side-effect of the procedures described in Section 5.1, `mac_prop_get` will return property settings in the driver.conf when no `mac_prop_t` structures are available, so that the "primary" interface (over which the diskless client boots) will continue to be able to use custom settings for the initial section of the boot process.

5 Fate of existing methods

This section discusses transition strategies from existing configuration methods. An overview of the "ndd compatibility" component of PSARC 2007/429 is provided in Section 5.2.

5.1 driver.conf

DDI properties are used by more subsystems than just network drivers, and there's a need to be able to define these properties through various means, so that `driver.conf` interfaces cannot be completely EOL'ed. However, in order to steer NIC drivers away from the `driver.conf` interfaces toward the `dladm` interfaces for "MAC properties", all of the following procedures will be implemented:

- The `mac_get_prop()` function described in Section 4.2 will look up a given `<prop_name>` by first looking up the available `mac_prop_t` structures, and if that fails, by looking up the available `ddi_prop_t` structures.
- Existing convention for documenting syntax accepted from the driver.conf files for network drivers is through comments in the template driver.conf file shipped with the installation. These comments will be appropriately modified to document the dladm invocation for setting the property.

Note that many `driver.conf` files are typically packaged as read-only by design, so that they are overwritten on install (see CR 6396173 "bge.conf is overwritten on upgrade/patch", CR 6430731 "e1000g.conf file removed on upgrade") which precludes handling of the case of carrying forward manual edits to the driver.conf files across an upgrade/install. In the cases where the `driver.conf` files are preserved, the class action scripts will be modified to accomodate changes in syntax accepted.

5.2 ndd

Even though `ndd` was originally created as a tool for managing the IP stack, it has been hijacked for the purpose of administrating device-driver properties. As a result, there is wide-spread usage (starting from at least Solaris 2.5.1) of using `ndd` to inquire/manage link-layer properties thorugh scripts used by System Administrators, and even via direct calls to NDD ioctls.

The "ndd compatibility" component of the umbrella case for PSARC 2007/429 will provide legacy support of existing ndd usage using the methods described below.

Drivers written after the delivery of the Framework component of PSARC 2007/429 will **not** have to support ndd, or provide entry points for the `ND_SET` and `ND_GET` ioctls. They should, instead, use the Brussels interfaces described in Section 3.1.

When GLDv3 drivers are converted to use the Brussels framework, existing support for ndd commands will be preserved by the following steps:

- Each driver converted to the Brussels framework will register the information about the **legacy `ndd` properties** with the mac layer. The information provided to the mac layer will include the name of the property (e.g., "`adv_autoneg_cap`"), min/max bounds on the values for the property, and optional pointers to private data to be passed down to the driver for setting the value of the property.

- For drivers converted to the Brussels framework, `ndd` calls will be intercepted and parsed by the mac layer for syntax, permission and bound-checks. Functions within the driver module will be invoked only if values private to the driver have to be set/read.

- Drivers not converted to the Brussels framework will continue to function in the pre-Brussels manner.

It is important to note that the Brussels project is the first step toward the larger effort toward ameliorating the deficiencies in ndd while providing better integration within SMF. Subsequent phases will re-evaluate and upgrade the usage of ndd in Layer 3, where we anticipate the conversion of many of the ndd tunables to recent Solaris features such as mdb and DTrace sdt probes.

5.3 kstat and ioctl calls

As with ndd, kstat output is frequently used to inquire about the current configuration of the driver, using, for example, the libkstat API, or in SNMP implementations. Thus, legacy support for kstat will need to be provided, at least, for the short term.

Some straight-forward cleanup is possible: the parameters tracked by `ndd` are frequently listed twice in the output of `kstat` (CR 6512220). For example, "`kstat bge`" duplicates `adv_autoneg_cap` information twice, both as `adv_cap_autoneg` under "`name: mii`" and as `adv_autoneg_cap` under "`name: parameters`". This duplication of output should be avoided. The MAC drivers introduce by Nemo [4] export many statistics that are documented in [6], however the stability level for these statistics is not clearly asserted as "Stable". By providing a standard "Stable" set of kstats for the Public properties, we can provide a single, reliable source for the statistic, and eliminate duplication.

Further, the current design can be made more elegant: requiring GLDv3 drivers to provide both the `mc_getstat` and `mc_{set, get}prop` entry points can be avoided, by unifying these entry points. In addition, analagous to the methods discussed in Section 5.2, the philosophy of intercepting and dispatching kstat calls in the mac layer can be adopted, so that drivers will not need to make any invocations to `kstat_create` and related functions.

An additional optimization area to be considered is that kstat collection under Nemo is expensive for drivers that need to do non-trivial effort to collect statistics. Currently, drivers that wish to avoid such expensive operations are required to manage their own periodic timers for determining the frequency at which hardware counters should be extracted. The GLDv3 framework can help optimize this process by providing information about the start/end of snapshots. Thus drivers can quickly and efficiently assess when to collect a snapshot (if needed), and when to return the device-specific resources associated with the snapshot back to the system. The driver/GLD interface would have one additional entry point, `mac_stat_snapshot()` that drivers could optionally provide with non-null values. The `mac_stat_snapshot()` entry point would be called at the beginning of kstat collection, and drviers could use it to shadow the stats from hardware to the device. The entry point could also be used to shadow individual properties. Thus, the kstats themselves would be handled by the framework which would implement the `ks_update()` routine, invoking `mac_stat_snapshot()` to query individual properties.

Note that, unlike ndd, the kstats themselves will not be marked Obsolete (though duplicate reporting of the same information will be eliminated).

The `ioctl(2)` system call is also used for driver configuration of some parameters like loopback mode (used by SunVTS), that could easily be treated as a Brussels property. The principles discussed earlier for cleaning up ndd and kstat may also be applied toward cleaning the ioctl handling. This aspect will be considered in Phase 2 of the Brussels Project.

6 Cleaning up the rest..

As each driver is converted to the Brussels framework, code cleanup and simplification should be attempted where possible. Some areas of cleanup are discussed in this section. Many drivers have a confusing range of properties that interact with each other in subtle ways. For example `bge` accepts all of the following as acceptable properties:

1. "`supported-network-types = "ethernet-10-half";`", which can be overridden by
2. "`transfer-speed=1000;`" which can be overridden by
3. "`speed=100; full-duplex = 0;`"

Each property has to be examined and duplicate methods such as those above should be simplified where possible (i.e., where no Stability commitments already exist).

7 Relation to MAC-type plugin architecture

As discussed in Section 3.2.3, the set of Public properties is logically subdivided based on link type, to reflect the fact that some public properties are only relevant to some link types (e.g., tunnel hop limit is not applicable to an ethernet interface, just as MII props are not meaningful to tunnel interfaces).

The MAC-type plugin architecture (PSARC 2006/248) was intended to provide an elegant way to implement linktype dependant features, and this architecture has been used for kstat management. A possible extension of the MAC-type plugin architecture for supporting MAC-type based Public properties could be achieved by requiring each MAC to provide a list of supported Public properties, each of which is identified by { property number, property name, property type, ...} during `mactype_register()`. The information thus registered would then be used by the framework for property string to property number mapping.

Extending the MAC-type plugin architecture in this manner has the advantage that there is no reliance on hard-coded mappings via `<sys/dld.h>`, and would allow dladm to quickly filter out invalid setprop/getprop attempts (e.g., an attempt to set tunnel encaps limit on ethernet interfaces). However this would require that libdladm upload the registered properties for each mac that starts up, requiring additional sytem calls for (minimally) the first setprop on the link. The additional system calls would also complicate the initial door call hand-shake between the driver and dladmd (See Section 4.2). In addition, since the property attributes needed for checking bounds and buffer management (`pd_check, pd_set, pd_get`) are managed within libdladm, using the MAC-type plugin architecture for defining Public properties would primarily be useful for doing the name to number mapping, and for assessing if a property is supported on the link type.

The complexity of the issues that arise hints at a flaw in the Nemo architecture itself: although the framework can handle pluggable MAC-types in the kernel, the links between this plugin framework and libdladm are not clearly defined. Improving the plugin framework will be considered in future Project Phases.

8 Acknowledgments

Thanks to the many people who have provided input including:

Eric Cheng, Garrett D'amore, Nicolas Droux, Randy Fishel, Roamer Lu, Peter Memishian, Erik Nordmark, Sebastien Roy, Rao Shoaib, Janlung Sung and the PAE team.

A Case study: MTU/SDU modifications for the bge driver

The Brussels framework provides more flexibility for changing the driver's mtu (typically used to get Jumbo Frame settings) than traditional methods via driver.conf. Thus the framework allows the user to

- allow persistent settings similar to the driver.conf mechanism, but without complex syntax to set per-instance values, and without requiring a reboot or update_drv procedure,
- use dladm's flexible syntax to view current settings,
- make "on the fly" modifications to properties when the driver is capable of absorbing the change.

However, in order to change the setting "on the fly", the driver has to be able to perform the reset operations associated with the change. In the case of bge, the mtu setting impacts various parameters like `jumbo_slots` that, in turn, may result in the resizing of Tx/Rx buffer sizes. These changes can only be implemented when the chip is stopped. Moreover, changes to the MTU on a registered mac will require an update to the `max_sdu` in the `mac_register_t`, and this change must, in turn, be propagated to DLS clients by sending `DL_NOTE_SDU_SIZE` notification in DLPI.

The constraints on driver state that must exist when attempting to change MTU are somewhat stringent in the case of the bge driver because changes to the MTU setting impact the way in which Tx/Rx rings are managed in the hardware. Other drivers (e.g., the tunnel driver) have more flexibility about modifying the MTU.

In the current prototype, when an attempt to modify the mtu is made to a started bge driver, the setprop settings are recorded in dladm's persistent repository. The change will take effect on the next stop/start of the mac, and a log message indicating this will be printed out by the bge driver.

B Example: implementation of private property

This section illustrates the steps involved in implementing a Private property, using the `bge` driver as an example.

It should be noted that Private properties should only be used with extreme discretion to handle exceptional cases where the driver needs to implement tunables to handle abnormal circumstances. The primary objective of the Brussels infrastructure is to emphasize the provision of common interfaces to all drivers, as opposed to customized, obscure tweaks for individual drivers.

The example considered in this section will demonstrate the conversion of the ndd parameter `drain_max` supported by `bge`. The `drain_max` parameter was introduced as part of the fix for CR 6416224 ("Single NIC bge Tx performance is very poor at 5000 connections") to handle hardware deficiencies in the Broadcom driver. The `drain_max` parameter affects the frequency at which hardware transmit is triggered under heavy stress. This parameter can take values between 1 and 512, and defaults to the heuristically determined value of 64.

The bge driver plugs into the Brussels framework by providing the `mc_setprop` and `mc_getprop` callback functions:

```
#define BGE_M_CALLBACK_FLAGS\
        (MC_RESOURCES | MC_IOCTL | MC_GETCAPAB | MC_SETPROP | MC_GETPROP)

static mac_callbacks_t bge_m_callbacks = {
        BGE_M_CALLBACK_FLAGS,
        /*
         * ...
         */
        bge_m_setprop,
        bge_m_getprop
};
```

which are then passed to the mac layer as part of `mac_register()`.

```
static int
bge_attach(dev_info_t *devinfo, ddi_attach_cmd_t cmd)
{
        bge_t *bgep;                            /* Our private data     */

        /*
         * ...
         */

        if ((macp = mac_alloc(MAC_VERSION)) == NULL)
                goto attach_fail;
        macp->m_type_ident = MAC_PLUGIN_IDENT_ETHER;
        macp->m_driver = bgep;
        macp->m_dip = devinfo;
        macp->m_callbacks = &bge_m_callbacks;

        /*
         * ...
         */

        /*
         * Finally, we're ready to register ourselves with the MAC layer
         * interface; if this succeeds, we're all ready to start()
         */
```

```
        err = mac_register(macp, &bgep->mh);
```

The callback function `bge_m_setprop` has been written to implement supported Public properties for the bge driver as shown below.

```
/*
 * callback functions for set/get of properties
 */
static int
bge_m_setprop(void *barg, void *parg, int pr_num, int pr_valsize, void *pr_val)
{
        bge_t *bgep = barg;
        char *pr_name;

        /*
         * pr_num identifies property number for public properties
         */

        mutex_enter(bgep->genlock);
        switch (pr_num) {
        case DLD_PROP_DEFMTU:
                /*
                 * updated the chipid.default_mtu
                 */
                break;

        /* .. */

        default:
                /*
                 * property number does not match as a known public
                 * property. See if it is a supported private property.
                 */
                pr_name = parg;
                err = bge_set_priv_prop(bgep, pr_name, pr_valsize,
                    pr_val);
                break;
        }
        mutex_exit(bgep->genlock);
        return (err);
}
```

Thus `bge_m_setprop` checks and handles each `pr_num` value that maps to a Public property, and handles unknown values of `pr_num` in the `bge_set_priv_prop()` function.

In order to support the `drain_max` tunable as a Private property, we rename it to `_drain_max` to meet the naming requirements prescribed in Section 3.2.4, and manage it as part of the `bge_t` structure for driver-private data. We now need a few simple changes to modify its value:

```
static int
bge_attach(dev_info_t *devinfo, ddi_attach_cmd_t cmd)
{
        bge_t *bgep;                            /* Our private data      */

        bgep = kmem_zalloc(sizeof (*bgep), KM_SLEEP);
        bgep->param_drain_max = 64;             /* initialize drain_max */

        /*
         * ...
         */
}

static int
bge_set_priv_prop(bge_t *bgep, char *pr_name, int pr_valsize, void *pr_val)
{
        int err = 0;
        long result;

        if (strcmp(pr_name, "_drain_max") == 0) {

                if (pr_val == NULL) {
                        err = EINVAL;
                        return (err);
                }
                (void) ddi_strtol(pr_val, (char **)NULL, 0, &result);
                if (result > 512 || result < 1)
                        err = EINVAL;
                else {
                        bgep->param_drain_max = (uint32_t)result;
                        /*
                         * reprogram chip, reset physical layer.
                         */
                        if (bge_reprogram(bgep) == IOC_INVAL)
                                err = EINVAL;
                }
                return (err);
        }
}
```

The ndd data-structure declarations for supporting `drain_max` may now be cleaned up. Specifically, the `nd_param_t` declaration for this tunable may be deleted, namely

```
-/* Performance tuning */
-{ PARAM_DRAIN_MAX,         1,    512,  64,     "+drain_max"            },
```

and replaced by the declaration in the `bge_t` data structure itself:

```
        /* ... */
        timeout_id_t            asf_timeout_id;
 #endif
+       uint32_t                param_drain_max;
```

```
 } bge_t;

 /* ... */

-#define        param_drain_max         nd_params[PARAM_DRAIN_MAX].ndp_val
```

C Public Properties

This section lists the tentative list of Public properties that will be introduced with the initial prototype. Note that new Public properties may be introduced at any subsequent time.

C.1 Physical Network Interfaces

C.1.1 MII properties

DESCRIPTION: All MII properties documented in `ieee802.3(5)` will be supported via dladm. In addition, `adv_10g_fdx` will be supported as defined in IEEE 802.3ae.

NOTES: The `ieee802.3(5)` page will be updated to reflect the support via `dladm`. Note that some properties such as `lp_cap_*dx` will be read-only properties.

C.1.2 ifspeed_clock

DESCRIPTION: Read-only access to link speed in Mbps

NOTES: See Section 3.2.5 for details of the `dladm show-ether` sub-command.

C.1.3 link_clock

DESCRIPTION: Applicable to gigabit twisted-pair links only. One side of the link is the master that provides the link clock and the other side is the slave that uses the link clock. Once this relationship is established, the link is up, and data can be communicated.

NOTES: Input values may be "master", "slave", "auto". [**?**]

C.1.4 link_default_mtu

DESCRIPTION: Set default mtu sizes in Bytes. If NIC supports jumbo frame, it could be set to values greater than 1500 Bytes. Valid range is [68 - 65536]

NOTES: Jumbo frames are not supported across all NICs. References: RFC 791 establishes the official minimum MTU of 68.

C.1.5 link_lance_mode, link_ipg0

DESCRIPTION: link_lance_mode and link_ipg0 are used to introduce additional delays, in addition to the delay setting by link_ipg1 and link_ipg2 (see Section C.1.6 for details).

If link_lance_mode is set to 0, link_ipg0 is ignored and no additional delay will be inserted. If link_lance_mode is set to 1, an additional delay will be added by link_ipg0. This delay by link_ipg0 is added on link_ipg1 and link_ipg2.

Valid input for link_lance_mode is 0 (Disable) or 1 (Enable). Valid input for link_ipg0 is in the range [0-65535]

NOTES: This parameter is typically used to reduce collisions in a CSMA/CD media, like a half-duplex ethernet hub env.

C.1.6 link_ipg1, link_ipg2

DESCRIPTION: link_ipg1 and link_ipg2 are basic parameters to adjust inter-packet gap (IPG) which is obtained as the sum of link_ipg1 and link_ipg2. Valid input range is [0-65535].

NOTES: The unit of link_ipg1 and link_ipg2 is *bytes time*, which is the time to transmit 1 byte on the wire.

C.1.7 flowctrl

DESCRIPTION: Valid input is one of {"no", rx, tx, "bi"}, where

rx: we receive, and act upon, incoming pause frames from peer
tx: we transmit pause frames to the peer, but ignore received frames
bi: bidirectional flow control, where we transmit and receive pause frames
no: no flow control enabled. We will not respond to, or send, pause frames

NOTES: The actual settings for this value are constrained by the capabilities allowed by the peer and the link-partner [**?**].

C.2 Tunnels

C.2.1 link_tun_hop_limit

DESCRIPTION: the maximum number of hops that a tunnel packet can travel from the tunnel entry-point to the tunnel exit-point.

NOTES: References: RFC 2473. These properties are currently being implemented by the GLDv3 version of the IP Tunneling device driver, which is under development by Project Clearview.

C.2.2 link_tun_encaps_limit

DESCRIPTION: the maximum number of nested encapsulations of a packet. Min 0, Max 255, Default: 4

NOTES: References: RFC 2473; These properties are currently being implemented by the GLDv3 version of the IP Tunneling device driver, which is under development by Project Clearview.

C.3 VNIC Properties

C.3.1 link_vnic_bw_limit

DESCRIPTION: Limits the full duplex bandwidth available on a Virtual NIC (to be delivered by Project Crossbow)

NOTES: References: flowadm(1m). These properties are currently being implemented by the Crossbow project, which is under development.

C.3.2 link_vnic_bw_guarantee

DESCRIPTION: If there is available bandwidth, the traffic related to this policy is allowed to exceed its guaranteed entitlement.

NOTES: References: flowadm(1m). These properties are currently being implemented by the Crossbow project, which is under development.

References

[1] "ethtool - Display or change ethernet card settings". http://www.linuxcommand.org/man_pages/ethtool8.html.

[2] "IBM-AIX documentation". http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp.

[3] "SAM- system administration manager". http://docs.hp.com/en/B2355-90692/sam.1M.html.

[4] PSARC 2004/571. "Nemo - a.k.a. GLDv3".

[5] PSARC 2005/132. "Clearview: Network Interface Coherence".

[6] Paul Durrant. "Nemo Interfaces Specification". PSARC 2004/571.

[7] PSARC 2006/623 PSARC 2006/406, PSARC 2006/517. "Wifi for GLDv3".

[8] PSARC/2006/406. "Extending dladm for WiFi: An administrative Overview". /net/sac.eng/export/sac/PSARC/2006/406/commitment.materials/dladm-wifi.pdf.

[9] PSARC/2006/406. "Libdladm APIs for managing link and secure properties". /net/sac.eng/export/sac/PSARC/2006/406/commitment.materials/libdladm.txt.