Project: Devname
Filesystem Driven Device Naming

Lucy Lai
yonghong.lai@sun.com
Solaris Core Technology Group

Table Of Contents

ABSTRACT

The Devname project builds the foundation for a simplified Solaris device naming model. The project implements an in-memory file system that exports the /dev namespace. This approach brings the flexibilities needed in exporting a subset of the /dev namespaces, intercepting individual /dev name lookup request, and storing a /dev name binding scheme into remote locations like a centralized name service backend.

The Devname project delivers the following components:

1. The *dev* File System – This file system exports the /dev namespace, supports multiple file system instances, and intercepts file system operations on /dev names.

2. Implicit Device Reconfiguration – This ~~obsoletes the reconfiguration boot and makes Solaris booting the same as other operating systems~~ feature provides an option for cold-plugged devices to be automatically configured into the /dev namespace without the need to do an explicit reconfiguration boot.

3. Simplified Zones Device Support - The project removes zones dependency on devfsadm [d], and encapsulates the device special file details inside the file system.

4. Simplified /dev/pts Namespace Implementation - This is the feature that demonstrates the directory based device name resolution architecture to achieve better system observability in local pty devices, simpler implementation, and a more reliable pseudo-terminal subsystem.

5. Directory Based device Name Resolution – This is the infrastructure provided by Devname to support flexible device naming scheme inside a /dev subdirectory. A plug-in mechanism is also provided for future delivery of directory based device naming scheme through kernel modules.

6. New Keyword for Name Services – This feature connects a file system device name lookup request with a name service database query request by providing a “devname” keyword. The keyword serves as a placeholder for all the name service varieties, with an initial implementation of NIS backend support.

These features are elaborated in the “Devname Architecture” section, with interfaces listed in the “Interface Tables” section. More details are supplied in the “Implementation” and “Appendix”.

1 Introduction

1.1 Background

Solaris devices are represented in two name spaces: /dev and /devices. The /devices namespace represents the physical path to a hardware device, a pseudo device, or a bus nexus device. It reflects the kernel device tree and is managed by the *devfs* filesystem (PSARC/2000/190). The /dev namespace contains logical device names used by applications. The names are either symbolic links to the physical path names under /devices or, in rare cases, device special files created via the mknod(1M) command or the mknod(2) system call. Most of the /dev names are automatically generated by devfsadmd(1M) in response to physical device configuration events. These naming rules are delivered by driver developers through link generator modules for devfsadm and entries in /etc/devlink.tab. It is also possible for system administrators and applications to create device special files and symbolic links directly, bypassing the devfsadm framework.

The global /dev namespace resides under the system root /dev directory. Some Solaris applications like ftpd create a chroot'ed environment and export a restricted subset of the system device names into its chroot'ed /dev directory. Solaris zones create virtualized Solaris instance and provide a subset of the system device names inside the virtualized /dev namespace.

1.2 Problem Areas

The current /dev implementation works fine in a static environment, but namespace changes are somewhat difficult to manage. The need to pre-populate the /dev namespace often results in complexity in handling device configuration changes. Also, the fact that no standard mechanism is provided today for exporting a subset of the system /dev names makes it very difficult and complex to manage the device names in a consistent way. Following sections describe the areas improved with the Devname initial delivery. The “Devname Future” section discusses the areas that can potentially be improved after Devname project is integrated into Solaris.

1.2.1 The Devfsadm Problems

Today, Devfsadm(1M), coordinating with link generators and /etc/devlink.tab, has been the standard way to update the global /dev namespace in response to kernel device autoconfiguration events. With the increasing need in the flexible device naming area [ref. CR #4096373, 4949283, 6325222, 4914135, 6311403, etc.], this static device naming approach does not adapt well. Several drawbacks have been noticed in the past:

- It does not allow user to rename a device. Once a name is assigned, it remains bound to the device at the bus location as indicated by the physical path, even if the device is replaced;

- It does not allow run-time device name re-binding. Both the link generator rules and /etc/devlink.tab rules are fixed when the software package is installed;

- It does not support network centric device naming. Devfsadm(1M) creates /dev names locally because the rules in either link generators or /etc/devlink.tab are local to the host. The same hardware may be named differently on two hosts, even when the device location is not changed. This is what happened to a tape drive when it is used in a SAN environment. There has been efforts in using the /etc/devlink.tab mechanism, which is able to get the same name for the same tape device. The approach still has some distance from what the customers asked for because the /etc/devlink.tab, and its contents, still live locally on each host. Maintaining a consistent copy of /etc/devlink.tab on all the networked hosts is not a trivial task;

- Only one single mechanism exists. This one-size-fits-all approach results in a very complex /dev name creation mechanism. The complexity involves many components, e.g. application issues ioctl to driver to create new minor nodes, which triggers a kernel sysevent transported up to syseventd in the userland which notifies devfsadmd. Then devfsadmd makes a libdevinfo call to get the minor node information from kernel, finds the appropriate link generator module for the device, and finally creates a new name in /dev;

- No tailoring option is available to create /dev names that is just a simple extension of their device minor name. For example, most of the pseudo devices are named as /dev/[minor-name]. In this case, the name can be easily created within the kernel. There is no need for a devfsadm link generator.

1.2.2 Reconfiguration Boot

By default, Solaris does not update /dev name space for new devices on a normal boot in order to reduce boot time. If new devices are added while a system is powered down, user must initiate a reconfiguration boot by explicitly issuing a special boot command, i.e. boot -r, or touch the "/reconfigure" file before the system is rebooted, in order to make the new devices visible in /dev. The need for reconfiguration boot is unique to Solaris and often confusing to new Solaris users.

One case in particular is perceived as a negative differentiator for Solaris. If a device is attached to the system when the system is powered off, Solaris typically won't notice such a device on reboot. Sysadmins must take an additional step, either rebooting -r or running devfsadm(1M) once the machine is up. No other OS requires this.

1.2.3 Local Zone /dev Namespace Management

In order for local zones to provide a subset of devices that is sufficient for basic OS functionalities to operate properly, several changes were made:

- The devfsadm(1M) utility was changed to be zone-aware. Zonecfg(1M) was made to be devfsadm-aware. The cross-references adds a lot of implementation complexity and increases development cost as well as maintenance cost.

- The zones /dev management is based on the mknod(1M) utility, which makes the

implementation depend upon device special files (i.e. dev_t). The drawback of this dev_t dependency is exposed the most when a non-native zone is implemented, or a local zone is migrated to a different host. The reason is that the same device may have different dev_t value on different host and different OS.

The motivation for reimplementing /dev in Zones are two-fold. One is to remove the Zone's dependency on devfsadm[d], simplifying the implementation. The other is to remove the dependency on device special files. This simplifies Zones migration (no need to worry about dev_t differences between one system and another). It also opens the door to removing specfs support from storage filesystems altogether.

1.2.4 /dev/pts Name Creation

The existing /dev/pts namespace implementation is complex, unreliable, has performance problems, and offers poor observability into the pseudo-terminal subsystem. The implementation relies on a hodge-podge of approaches to keep /dev/pts working. The following are the approaches taken in Solaris. Each of the approach reveals some problems that could be solved by the *dev* filesystem reimplementation. The first approach is to pre-create /dev/pts nodes before the corresponding pseudo-terminal is created, to overcome the asynchronous nature of sysevent mediated link generation. The second approach (which renders the first superfluous) involves taking a snapshot of the kernel device tree and creating/checking every /dev/pts link in the system before opening a /dev/pts node. This can create performance issues during pseudo terminal creation on large systems. Neither approach will work if the devfsadm daemon dies for any reason. Finally, /dev/pts is simply a persistent collection of symlinks that has been created in the past. The symlinks do not reflect current system state. In fact, since symlinks can be pre-created, they may represent pseudo-terminals that do not exist. The situation is particularly confusing with zones, since all (potential) pseudo-terminals in the system are reflected in all zones, when only a subset of these are available for a zone.

2 Interface Tables

2.1 Public Interfaces

Interface	*Commitment Level*	*Comments*
dev	stable	Devname filesystem type

2.2 ON Consolidation Private Interfaces

Interface	*Comments*
mount_dev(1M)	Command to mount an instance of Devname Filesystem
di_prof_t	Opaque handle to a non-global filesystem instance profile datastructure
di_prof_init	Allocate and initialize a di_prof_t
di_prof_fini	Free a di_prof_t
di_prof_add_dev	Include a list of device or directory to the profile
di_prof_add_exclude	Remove a list of device or directory from the profile
di_prof_add_symlink	Add a symlink, with source and target, to the profile
di_prof_add_map	Add a device remappind, with origin and new, to the profile
di_prof_commit	Commit the profile to Devname filesystem
/kernel/devname	Devname plug-in module repository directory

2.3 Project Private Interfaces

Interface	*Comments*
sdev_node_t	Devname filesystem node datastructure
/etc/dev/devname_master	The master configuration file for /dev sub-namespaces

Interface	*Comments*
/etc/dev/README	Readme file for /etc/dev directory
devname	Keyword for /etc/nsswitch.conf
/kernel/devname/devname_nsconfig_mod	Plug-in module for filesystem and userland name service communications
di_devname_ns_setup	Establish the connection between dev and name service switch subsystem
di_devname_get_mapinfo	retrieve the named map entries
di_devname_get_mapent	retrieve a named entry in the map
di_devname_matchnode	Match a minor node property
struct devname_ops	Directory ops implemented in a plug-in modules
DEVNAME_NS_PATH	Physical pathname protocol
DEVNAME_NS_DEV	/dev pathname protocol
mod_devfsops	Devname plug-in module mod_ops structure
devname_handle_t	An opaque handle for a devname sdev_node
devname_get_vnode	retrieve the vnode from a handle
devname_get_path	retrieve the node path from a handle
devname_get_name	retrieve the node name from a handle
devname_get_dir_handle	retrieve the parent directory's devname_handle from a child handle
/etc/devices/devname_cache	Filesystem private data
finddevhdl_t	libdevinfo typedef
device_exists(3DEVINFO)	libdevinfo API
finddev_readdir(3DEVINFO)	libdevinfo API
finddev_by_name(3DEVINFO)	libdevinfo API
finddev_close(3DEVINFO)	libdevinfo API
finddev_next(3DEVINFO)	libdevinfo API

3 Devname Architecture

The Devname project provides an enabling technology that can be utilized in future projects to provide simplified and improved Solaris device naming features. The new architecture employs filesystem operations as well as customized filesystem instance mounting operations. As initial enhancements to Solaris, the project eliminates reconfiguration boot, simplifies zones device support, and provides an improved /dev/pts implementation.

3.1 The *dev* File System

The *dev* file system provides a uniform interface to logical device names, i.e. /dev files. The primary purpose of *dev* file system is to intercept lookup calls to /dev files and translate them to calls to /dev subdirectory specific name resolution routines, or the default if no special routine is provided.

The *dev* file system is mountable in a restricted way. As to be described in section 3.2, the global instance is mounted onto /dev early when the system is booted. Other mounting operations occur when a subset of the global /dev names are exported into chroot'ed /dev environment. While the global instance is mounted by the kernel, other file system mounts is are normally initiated in a subsystem program like zoneadm(1M).

The in-memory file system states can be persisted across system reboots. In order to persist the changes of some file system states, e.g. file permissions, ownerships, ACLs, local backing store(s) are used. By default, the backing store is located at the same directory as the file system mount point. An alternate backing store location can be specified through the mount options. To accommodate the volatile nature of some device names, e.g. /dev/pts nodes, the file system also allows a /dev subdirectory to choose not to persist anything into the backing store. Such a directory is called dynamic directory.

3.2 The *dev* File System Multiple Mounting

The *dev* file system controls all the /dev logical device names, including the ones inside the system root /dev namespace and any existing ones inside the chroot'ed /dev namespace(s). The key here is to mount an instance of the file system onto each /dev namespace, while differentiating the mounting of the first file system instance, called global instance, from the rest of the file system instance(s), called non-global instance(s).

3.2.1 The Global Instance

Early when Solaris is booted, a *dev* instance is instantiated. This instance is mounted directly onto the system root /dev directory. It is different from other instances in that

- It provides the master set of device names on the system. The filesystem is strictly mounted onto the root /dev directory. All the existing on-disk names are transitioned into the file system transparently. The on-disk /dev namespace is also used to persist

the global instance state.

- Hotpluggable hardware device names are first created in the global instance before they can be exported into other instances.

- Most of the pseudo device names are first created in the global instance. Exceptions are the volatile device nodes inside a dynamic /dev subdirectory as to be discussed in section 3.2.3.

- An existing device name goes away from the global instance first. Then it disappears from the chroot'ed /dev namespace. Exceptions are the dynamic nodes.

- New /dev nodes may be created by mknod(2), symlink(2), mkdir(2), open(2) (with O_CREAT flag set).

- Existing /dev nodes may be deleted by rm(1), rmdir(2), unlink(2), or renamed by rename(2).

- Global instance is always mounted. The "mount -p" shows this entry for the *dev* file system: "/dev - /dev dev – no".

3.2.2 The Non-Global Instance(s)

After the global instance is successfully instantiated, one or more *dev* instance(s) can be mounted by applications like zoneadm(1M), or ftpconfig(1M). These non-gobal instances can be unmounted anytime, for example, when tearing down an anonymous FTP environment or halting a local zone. When such a instance is mounted into local zone, named zone2, the "mount -p" shows this *dev* instance as:
"/dev - /export/home/zone2/root/dev dev – no".

Unlike the global instance, these instances are instantiated with an empty /dev namespace. Except for the dynamic directories, the chroot'ed /dev namespace only contains a strict subset of the device [re]names in the global /dev namespace. The subset of devices is represented by a list of matching rules, called a profile. The profile is compiled by the system administrators that is configuring the application. Upon successful mounting a *dev* instance, the mounting application passes a profile down to the file system in the kernel. From then on, the file system is responsible for populating each instance according to the associated device profile. The file system also allows dynamic updating of the device profile.

Populating each non-global instance occurs on-demand within the file system. Initial population happens when the first device in the profile is accessed (i.e. looked up), in which case the /dev [sub]directory is fully populated with all the profiled devices for the [sub] directory if the device is existing in the global instance. New devices may show up later when the corresponding device name is configured into the global instance. Already populated device name may be removed from the directory when it is accessed and the file system detects the absence of this device name from the global instance. For example, a hardware device is removed from the system.

Sometimes, a global device may have a different name in the non-global instances. This is handled by the renaming rule in the profile. A renaming rule needs to provide the original device name for a profiled device. Thus the file system is able to check the existence of the original device name in the global instance and populate the profiled device with the device attributes of the original device.

3.2.3 Dynamic Directory

Some directories contains session-oriented device names that are dynamically created when a session is initiated and destroyed when the session is closed. The /dev/pts nodes are examples that are volatile in terms of their lifetime on the system.

The file system treats dynamic directories specially. Normally, this kind of directories are associated with its own special device name resolution routines. As to be discussed in section 3.3, the device name creation and destruction are determined through the mechanism implemented in the special routine. Thus, it is up to the mechanism to characterize the device names in a global instance versus non-global instances. They no longer have the subset relationship as discussed in section 3.2.2.

The /dev/pts reimplementation as to be discussed in section 4.3 is a good example of such a dynamic directory.

3.3 File System Intercepting of Device Name Lookups

3.3.1 The *dev* File System Device Name Lookups

The *dev* file system provides a dynamic environment that can be deployed to solve most of the problems listed in section 1.2. The key is the middle-man position of the file system that intercepts all the /dev device name lookups, i.e. vop_lookup(), which is searching the /dev or a /dev subdirectory for a path-name component matching the supplied relative path-name string. The *dev* file system allows a customized name resolution mechanism to be registered for individual subdirectory and applies the customized name resolution mechanism to resolve all the path-names inside the same subdirectory.

Figure 1, Device Name Resolution with Devname

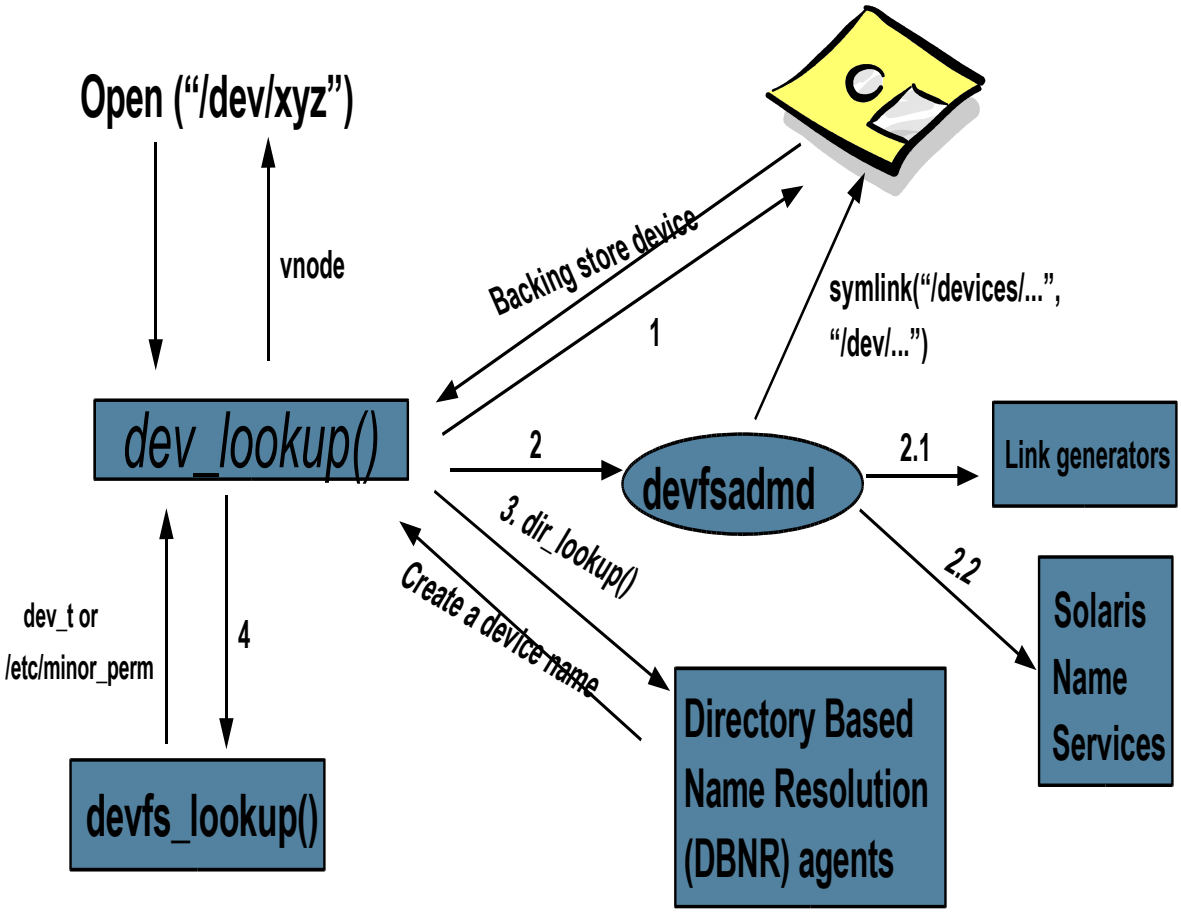

Figure 1 illustrates how the *dev* file system vop_lookup() works to handle co-existing of the device name resolution mechanism varieties. Note, the Devfsadm mechanism exist today and continues to work with the new architecture. Additions are the DBNR agents, which

provides the directory specific customized name resolution, Solaris name services, which provides the centralized storage of device name binding scheme, and in-kernel *devfs* file system interaction, which extends the dynamic kernel device tree configuration feature to the /dev layer.

3.3.2 Directory Based Device Name Resolution

The *dev_lookup()* is the entry point that resolves a /dev path-name according to its residing directory. Like the rest of the other *vfs* based file systems, searching a path-name component string starts with the top of the file system, i.e. the /dev directory, continues to the next layer of the /dev subdirectories, and so on. By default, resolved /dev names are also cached in the memory organized similar to the on-disk file system hierarchical structure. Unlike other *vfs* file system, *dev_lookup()* is implemented to be smarter than just searching the directory cache and looking up the backing store for the path-name. The *dev* file system keeps an internal switch table, which associates a /dev subdirectory to a vectored routine that implements its individual name resolution scheme. If available, the vectored name resolution scheme is applied to resolve all the path-names inside the subdirectory.

To gain device [re]naming flexibilities, a Solaris subsystem can deliver a customized name resolution routine into the *dev* file system switch table. The Directory Based Name Resolution, abbreviated as DBNR, dispatching works well when the /dev subdirectory is a collection of all the device names that represent the same type of devices which follow the same device naming scenario. Existing /dev namespace exhibits such a tendency, which contains a group of subdirectories with each representing a different type of device. For example, /dev/rmt represents all the tape devices on the system. It is worth mentioning that current /dev namespace doesn't have subdirectories for all the Solaris supported device classes. In practice, a project normally creates a /dev subdirectory to host all the devices that are interested to the subsystem the project is supporting. For example, the Solaris network vanity names project is creating a /dev/net subdirectory to host all the VLAN devices on the system, and a *net_lookup()* routine can be vectored into the *dev* file system while keeping the internal implementation of dynamic VLAN device renaming inside the ClearView subsystem.

There are several advantages to decoupling the device naming scheme from the file system device name resolution framework, i.e. *dev_lookup()*. Listed here are,

1. The default file system lookup routine, i.e. *dev_lookup(),* can be simple and generic.

2. The DBNR routine, i.e. *subdir_lookup(),* can be tailored to fit the device owning subsystem needs.

    - The /dev/pts nodes are as simple as appending the pty slot name to the "/dev/pts" subdirectory name. This does not need the complex devfsadm approach at all. Section 3.6 discusses the details to simplify the implementation.

    - The /dev/net VLAN devices. The ClearView subsystem needs a way to dynamically create VLAN devices according to the physical NIC devices hotplug

events and sysadmins dladm(1M) network device renaming commands. A customized *net_lookup()* can implement the needed dynamic features while keeping the details transparent to the rest of the /dev namespace. The final delivery of this functionality is going to happen when the ClearView project is also integrated into Solaris. Next section discusses the plug-in mechanism provided by devname to facilitate such an effort.

- The networked tape device naming. In a shared SAN environment, the same tape device needs to be named the same on all the hosts so that the backup software can be configured consistently across the enterprise. The existing solution is to put an identical copy of /etc/devlink.tab on all Solaris hosts. This is not convenient and easy to make mistakes. It will be ideal to put a single copy of the /etc/devlink.tab on a centralized name server. Section 3.4 discusses the architecture to connect a /dev name lookup request with a name service database query.

While it is out-of-the-scope of this project to architect fully populated /dev sub-namespaces, the project expects the DBNR infrastructure to be deployed when a /dev subdirectory seeks to provide the end user flexibilities in device [re]naming.

3.3.3 DBNR Plug-in Mechanism

The DBNR plug-in mechanism is developed to facilitate future projects with a convenient method to implement device type specific (re)naming semantics on top of the existing link generator mechanisms. The module is designed in such a way that the filesystem internals are not prerequisites for writing such a module, at the same time the module can have full control of how a device name is created and renamed.

3.4 Interaction with Name Services

3.4.1 Devname and Name Services

The *dev* file system provides the opportunity to associate a /dev subdirectory with individual naming scheme. Thus, it is possible to have a /dev subdirectory that all the names can be resolved through contacting a name service on which stores all the device name binding rules. For example, the local /etc/devlink.tab copies can be moved into a name service database. Local host retrieves a named device entry from the name service database. The retrieved entry is guaranteed to be consistent across all the hosts.

In order to do this, following components are involved when the *dev* file system intercepts a device name resolution request:

- The /dev subdirectory. From the implementation perspective, a dedicated subdirectory is recommended. For example, /dev/tape can be used to host all the name service named tape drives. In the case of mixing the name service named tape drives with the devfsadm named tape drives, extra work is needed to enhance the tape link generators to handle two types of tape names. Following discussions focuses on a dedicated subdirectory for the name services named devices.
- The *subdir_lookup()* routine that transports the file system *vop_lookup()* requests to userland name services, and creates the file system node upon returning from a successful name service query. The project implements a generic name service lookup routine, i.e. *devname_nslookup()*, through a plug-in module. By default, this module is not loaded until a /dev subdirectory is configured to be using any of the name services.
- The userland process that receives the *devname_nslookup()* requests from the kernel, dispatches the lookup requests to the configured name service subsystem, i.e. /etc/nsswitch.conf, and transports the name service query results back down to the kernel. In order to do this, devfsadmd(1M) is enhanced to support a door for the communication between the *dev* file system and devfsadm daemon.
- The name binding rules to be stored in the name services. Similar to devfsadm link generators, these rules are specific to device types, and provided by the device supporting subsystem. The project is not intended to make assumptions about a particular type of device naming scheme. Delivering the end user benefits should come in the future projects that defines the device type specific name binding scheme, defines device name creation protocols between the binding rules and the in-kernel device name creation inside *devname_nslookup()*, and provides easy to use sysadmin tools.

3.4.2 Devname NIS Example

Devname project delivers a “devname” keyword into /etc/nsswitch.nis for NIS clients. The purpose of this feature is to proof-of-concept the centralized device naming architecture. As discussed in section 3.4.1, Devname expects future projects deliver the two missing parts, with one as the device type specific naming scheme, and the other one as the support of all the Solaris name services and configuration tools.

4 Devname Implementations

4.1 Implicit Device Reconfiguration

4.1.1 Implicit Device Reconfiguration

The *dev* filesystem provides us with a unique opportunity to improve the configuring new device names experience. The filesystem can invoke devfsadmd(1M) to resolve the lookup operation of a non-existent device name or to complete the readdir of a /dev directory. Thus names for new devices can be configured without the need for a special reconfiguration boot.

In practice, all /dev names required to complete the preliminary phase of boot, mounting root, usr and swap for example, and pseudo devices such as /dev/null, are set up and configured as part of the install process. System daemons, such as vold(1M) and picld(1M), explore what's available on a system as they start up, and some of this involves searching for a variety of devices, some of which do not and may never exist on a given system. For example, some platforms provide memory controller devices in /dev/mc, referred to by picld (1M). Another example of an attempt to discover a non-existent device is /dev/NULL by dtlogin, for what purpose we can only guess. While we can fix some such bugs (see 6407277), in general it is obvious that real-world software is very cavalier with respect to / dev, and that without a mechanism to avoid it, an implicit reconfig would be performed quite early during each boot.

For this purpose, a cache of unsuccessfully looked up device names, referred to as negative cache, during system boot and after an implicit reconfig is accumulated and persisted. This gives the /dev filesystem a working set of device names for which implicit reconfigure need not be initiated.

4.1.2 Persisted *dev* File System Negative Cache

The negative cache is persisted in /etc/devices/devname_cache. This is the same directory used for the devid, mdi and devinfo snapshot caches. By persisting the data, the negative cache entries recorded during the most recent reconfigure boot are available to subsequent boots.

The persisted data can be recreated if the persistent store is removed or corrupt, as the persistent store is updated for non-existent devices accessed during both reconfigure and non-reconfigure boots. No administration of the persisted store is necessary and in practice it is expected that this data should rarely, if ever, need to be updated once established.

Since the negative cache initial state is not required in order to mount root, the persistent store may be read at the time the *dev* file system is mounted. This implies that the persistent store file does not need to be included in the GRUB boot archive, either.

The effect of an entry in the negative cache is only to avoid triggering implicit reconfiguration for lookup of a node or directory not otherwise discovered. A name is added to the negative cache when a implicit reconfiguration operation is attempted and the device name is not generated. A name in the negative cache is removed if that device name is

created, via for example making a directory, a device node or a symbolic link.

In practice, after a couple of attempts at implicit reconfiguration, a given system's negative cache typically reaches steady state and seldom changes.

To explain why the negative cache does not need explicit management, let's explore some possible scenarios. Consider the case that some driver hasn't yet been installed on a machine and the user runs something to access it, say ls -l /dev/xyz. The lookup would invoke reconfig, xyz would not be discovered and would be entered in the negative cache. Then the user installs the package with the necessary driver, or perhaps gets a corrected version of the driver supporting the new device. The install/update process should invoke add_drv, update_drv or devfsadm, and and either install a new/updated driver.conf file, a devfsadm link generator module or create the name directly via mkdir, link or mknod, any of which would remove the name from the negative cache as part of the create operation.

The public means of correcting an invalid entry in the negative cache for a name created by a link generator would be simply to invoke devfsadm. For example, attempting to lookup a newly attached but powered-off disk could put that device name (if one guessed correctly) in the negative cache. After powering the device on, the admin should invoke devfsadm, which is our documented procedure today.

For anything not covered by a link generator, well, it would be a bug that the name hadn't already been created directly. Further, the only meaning attached to an entry in the neg. cache would be to not attempt implicit reconfig; but implicit reconfig, invoking devfsadm, cannot create something not covered by a link generator anyway. So the way to correct a missing /dev/xyz name not created by a link generator and not created by the xyz install process due to a bug would be a manual operation (mkdir, link, mknod).

There is one other mechanism that an install procedure could use which is /etc/devlink.tab. But that's covered by devfsadm so for our purposes that's equivalent to the link generator mechanism.

The negative cache is disabled under following three situations:

- When an explicit reconfiguration boot (boot -r and /reconfigure) is requested;
- When the SMF multi-user milestone is reached;
- When the negative cache size reaches a maximum size.

4.2 Zones Device Support

4.2.1 Zones Device Support Background

Zones provide a subset of devices to applications, enough for basic OS functionality to operate properly. A fixed set of pseudo devices are always present in a Zone (e.g. dev/null, dev/zconsole, dev/pts). Additional devices may be exported at the discretion of the global zone administrator. The mechanism for exporting devices is to run zonecfg(1M) and add matching rules based on /dev names. The rules are persisted in an xml file, to be processed by devfsadm during Zone creation/setup.

/dev is readonly to local Zone users. The content is updated in several cases:

- A new pseudo terminal may be created in response to a new login request. This was handled by a door call to devfsadmd in the global zone.
- If physical devices are exported to a local zone, new device entries may come and go as physical devices are added and removed. This was handled automatically by devfsadmd in the global Zone.
- Device allocation in Trusted Extensions automatically exports removable media to Zones matching the security label on the media. This was done in device allocation software via mknod from the global zone.

4.2.2 Devname Implementation

Device service in Zones is supported by the ability to instantiate multiple filesystem instances.

To create a new instance at /zone/dev, simply mount the filesystem on it. As discussed in section 3.2.2, the /zone/dev is empty when it is first mounted. Later at run time, the /zone/dev is populated with the profiled devices.

The device profile for each local zone instance is defined in the zone's xml configuration file. The xml file contents are parsed by zoneadmd(1M) which passes the processed device list, or profile, down to the *dev* file system. The file system then associates the profile with the mounted local zone instance and populates the local zone /dev namespace according to the device names in the profile.

When a device is accessed the first time, the filesystem checks the matching rules defined in the profile and, if matched, performs an in-kernel lookup of the corresponding name in the global instance. If the device exists in the global zone, a new vnode is created in the local instance and returned to user.

When a hotplug event happens in the device tree, the kernel updates a generation number. Each local filesystem instance checks the generation number and updates its name space as needed. The creation of new pseudo terminals in Zones are handled as described in section

4.3, because the /dev/pts is a dynamic directory.

The management of device access attributes works as follows. When a Zone is first booted, device permissions are set to the system default as defined by /etc/minor_perm. When a user performs chmod, chown, or setacl, the filesystem creates a shadow node to persist the attributes. The shadow node is kept even after a Zone is shutdown. Zones migration code can look at the shadow nodes for the set of devices with non-default access attributes.

4.2.3 Implementation Changes

As part of this project, zoneadmd and libzonecfg are modified to call interfaces in section 2 and provide Zone device support via the new filesystem. Existing devfsadm code for handling zones is completely removed. It is worth mentioning that these changes are limited to the internal implementations. No user visible changes are expected.

4.3 Dynamic /dev/pts Name Creation

By utilizing the DBNR feature of devname, a simpler and more reliable implementation of /dev/pts is achieved.

4.3.1 The /dev/pts Namespace Background

The pty subsystem consists of two drivers: ptm and pts. The pty subsystem simulates a terminal, with the master (ptm) representing the terminal (with its terminal controller) and the slave (pts) representing the host/OS. Creation of a pseudo terminal begins with the open of the master side node /dev/ptmx. This is a self cloning node which does the following:

1. allocates a minor for the new pty.
2. allocates a slot in the pt_ttys array for the new pty. A pt_ttys structure is the data structure associated with a pty.
3. If there are no free slots in the pt_ttys array we grow the array to two times the current maximum (up to a certain size beyond which the growth is fixed).
4. In addition to the pt_ttys array, the pty subsystem also creates slave minor nodes so that slave device special files (/dev/pts) are available for consumers to open.

Once the master open succeeds, client software has a file descriptor representing the newly allocated minor. Minor numbers are identical between the master and the slave side. The client software then performs the following steps:

5. The client invokes grantpt(). This uses a setuid program (pt_chmod) to change permissions on the slave device nodes in userland.
6. The unlockpt() function is then called which unlocks the slave minor node for open.
7. ptsname() is invoked to determine the name of the /dev/pts file to open.
8. The client then opens the slave device file.

The client now has both ends of a full duplex connection for use as a pseudo terminal.

4.3.2 ~~Devname Implementation~~The Problem:

~~The /dev/pts node creation is dynamic, through following steps:~~

1. ~~User does a lookup of (say) /dev/pts/1 after a new pty is successfully allocated in the pt_ttys table.~~
2. ~~The VFS layer parses each component of the pathname, invoking the lookup routine of the corresponding filesystem.~~
3. ~~When processing the final component (1), a /dev/pts specific lookup routine is invoked.~~
4. ~~This routine consults the pty subsystem to determine if this is an allocated pty.~~
    1. ~~If it is, a character device special file is created in /dev/pts and the corresponding specfs vnode is returned to the caller.~~

From the perspective of /dev/pts node creation there are several problems:

1. In Section 4.3.1 (4), the pty subsystem relies on sysevents to propagate minor node

creation events to userland. Once these events reach userland, the devfsadm daemon (devfsadmd) creates the corresponding /dev/pts entries. Unfortunately the sysevent framework is asynchronous. The events may reach userland some time after the open on the master node has completed. There is the very real possibility that the open of the slave /dev/pts entry fails because the entry hasn't been created yet. To overcome this problem, the /dev/pts subsystem pre-creates minor nodes i.e. if the current pt_ttys array is of size N, it creates 2*N minor nodes. The expectation is that by creating minor nodes ahead of time, by the time the need for that minor node arises, the corresponding /dev/pts minor node will have been created. Unfortunately there is no guarantee this scheme will work in a busy and overloaded system.

2. PSARC/2000/310 introduced the di_devlink_init() interface which allows programs with appropriate privileges to programatically request the devfsadm daemon to create /dev links. The link creation can be restricted to that of a specific driver. The pty subsystem uses the setuid program pt_chmod (see 5 above) to invoke di_devlink_init() during ptsname(). This ensures that the link is always present for an open. This renders the 2*N pre-creation of minor nodes redundant, but that code was never removed. One drawback of this approach is that it penalizes every /dev/pts open - a full snapshot of the device tree and creation/checking of ***all*** /dev/pts entries has to happen before a /dev/pts entry can be opened.

3. The current /dev/pts entries are static and reside typically on a ufs filesystem. The pty subsystem is non-persistent i.e the pt_tty array is reallocated from scratch on the next reboot. However the /dev/pts entries are persistent. If the pty subsystem grows in size, the number of /dev entries grows twice as fast and then persists across reboots. The links in /dev/pts are never deleted, cluttering the /dev/pts directory.

4. The /dev/pts subdirectory offers no observability into how many ptys are active in the system. It is simply a static collection of links that have been created at some time in the past. This is now a bigger problem with the introduction of zones. With zones, there is the notion of pty ownership. A pty is owned by exactly one zone - either a local zone or the global zone. The global zone cannot open a local zones /dev/pts entry nor can a local zone open a /dev/pts entry allocaated to the global zone. Unfortunately, all /dev/pts entries are replicated in all zones (both global and local). Not only is this a poor situation from the observabilityperspective, it is potentially confusing.

4.3.3 The Solution

With the introduction of devnamefs we can intercept lookups in /dev. This solves most of the problems listed above. In addition, we will use this opportunity to fix the remaining problems in /dev/pts.

1. Node creation: Since the /dev filesystem is dynamic there is no need to pre-create /dev/pts links. A lookup in /dev/pts can be intercepted and the corresponding /dev/pts node can be created on demand. The slave driver (pts) no longer invokes ddi_create_minor_node(), nor are symbolic links created in /dev/pts. Only the corresponding character device files are created by the /dev filesystem.

The following happens during a lookup in /dev/pts:

User does a lookup of (say) /dev/pts/1. The VFS layer parses each component of the pathname, invoking the lookup routine of the corresponding fileysystem. When processing the final component (1), a /dev/pts specific lookup routine is invoked. This routine consults the pty subsystem to determine if this is an allocated pty. If it is, a character device special file is created in /dev/pts and the corresponding specfs vnode is returned to the caller.

As a result, both the sysevent and devfsadm frameworks are not involved in node creation. All of the 2*N node pre-creation logic can now be removed from the pty subsystem. The lookup of a /dev/pts entry for an allocated pty is now guaranteed to succeed.

2. di_devlink_init(): This is no longer needed to guarantee node availability. This will significantly improve the performance of pty processing.

3. Non-persistence: ptys are reallocated after each boot of the system i.e. they are not persitent. With the devname project, the same behavior applies to /dev/pts links. The /dev/pts nodes will not be persistent and will be in-core entities only. Not only does this improve performance during pty allocation but it also prevents large pty allocations from cluttering up the /dev/pts directory.

4. Observability: Only allocated ptys are now visible in /dev/pts. This greatly improves visiblity into the pty subsystem. In addition each zone, (global or local) will only display the ptys allocated to that particular zone. This removes any potential confusion and security concerns users may have regarding pty allocation and local zones.

5. Removal of pt_chmod: As before, we use the grantpt() interface to change the owner and group id of the slave device node. However, we no longer use the setuid program pt_chmod. grantpt() determines the real user ID of the calling process and the group id for the "tty" group and sends them via an I_STR ioctl to the master driver (ptm). ptm records these IDs in the slot table entry for that pty. When the pts slave node is instantiated by the /dev filesystem, it uses these values to assign the correct owner and group for the /dev/pts entry. The /dev filesystem allows us to remain fully compatible with existing grantpt() behavior while getting rid of pt_chmod.

4.3.4 Summary of Changes

To summarize, the devname filesystem makes the following changes to the /dev/pts implementation:

1. No reliance on sysevent framework

2. The slave driver (pts) no longer creates minor nodes

3. devfsadm no longer creates /dev/pts entries. The devfsadm link generator for pts is

removed

4. di_devlink_init() is not used by grantpt()
5. /dev/pts entries are no longer persisted
6. The /dev/pts entries reflect instantiated ptys rather than being a collection of symlinks many of which are dead. /dev/pts entries in a zone reflect only those ptys allocated to that zone.
7. pt_chmod is removed. The /dev filesystem applies the correct owner, group and mode entries.
8. A /dev/pts specific filesystem lookup routine is provided by the /dev filesystem.

4.3.3 ~~Implementation Changes~~

~~The following changes have been made to the pts subsystem:~~

- ~~the pts link generator rule is removed;~~
- ~~pts minor nodes are removed;~~
- ~~a pts-specific lookup logic is implemented;~~
- ~~pts nodes are no longer persisted;~~
    - ~~pt_chmod dependency is removed. The user credentials are generated dynamically when the /dev/pts nodes are created.~~

4.4 Devname NIS Support

When we talk about NIS name services, a picture jumps into the sight with the NIS server, NIS client, /etc/nsswitch.conf file, and of course, the software that is using the database stored on the server. Normally, these pieces fit together when the software needs a piece of information stored in the database. What happened is the software initiates a request to the NIS client, in turn the NIS client gets the requested item from the remote server and pass the result back to the software.

The "devname" keyword implementation follows the same scenario. This is how each piece looks like in the world of "devname":

- NIS server - no implementation changes other than adding the map databases
- NIS client (or the /usr/lib/nss_nis.so.1 library) - no implementation changes
- name service switch - no implementation changes other than adding the new "devname nis" entry on NIS client hosts
- Devname subsystem- This is where the "devname" keyword is implemented and the master configuration map, i.e. "devname_master" and directory configuration maps, e.g. "tapes", information are consumed.

Since there are no implementation changes in other components than devname subsystem, let's discuss the devname subsystem in more detail.

The devname subsystem contains following pieces from the perspective of being a NIS client consumer.

4.4.1 Libdevinfo Library Interfaces

Listed here are the new APIs developed for userland applications, in this case, the Devfsadm daemon, to establish the communication with name service and retrieve map entries from a NIS database. They are:

di_devname_ns_setup - establish the connection between the application and name services
di_devname_get_mapinfo - Retrieve all entries from the named NIS map
di_devname_get_mapent - Retrieve a named entry from the named NIS map

Note, these APIs are implemented in such a way that all the name service communication details are hidden from the calling applications. The NIS APIs are called within the di_devname_xxx interfaces.

Another note, these APIs are designed to be extendable to work with other name services, like LDAP. The reason is that the libdevinfo APIs keep an internal switch table. Future projects can implement a LDAP routine and vector the LDAP routine into the switch table.

When the system is configured as a LDAP client, these APIs automatically vectors into the LDAP routine and calls into LDAP interfaces accordingly. This second layer of name service switch, if the /etc/nsswitch.conf is the first layer, is transparent to the API callers.

4.4.2 Enhanced Devfsadm Daemon

As discussed in section 3.3.1, Devfsadmd is functioning as the bridge between the userland name services and the kernel *dev* filesystem.

In order to do this, Devfsadmd creates a new door. The purpose of this door is to handle the name service lookup requests from the *dev* file system. When the door is created, Devfsadm daemon establishes a communication with the name service switch through the di_devname_ns_setup library call.

Upon receiving a name service lookup request, Devfsadm daemon fetches the named entry or the named map by making the di_devname_get_mapent or di_devname_get_mapinfo library calls.

In the end, Devfsadm daemon passes the results from the NIS database down to the *dev* file system through the door_return() calls.

4.4.3 Kernel Components Requesting Name Service Lookups

As discussed in section 3.4, a name service lookup routine, i.e. devname_nslookup(), is implemented to handle the communication between userland devfsadm daemon and kernel file system. A /dev subdirectory may choose to store all the naming scheme in a NIS map. Thus, the devname_nslookup() will be invoked to resolve a device name in such a directory with following steps:

The name lookup request travels to userland devfsadmd that calls the libdevinfo APIs, in which the name lookup request is translated into a NIS database qurey. The query is then passed to the NIS subsystem on the local host, which forwards the query to remote NIS server. In the end, the name binding information stored in the database is retrieved and traveled the opposite direction to eventually comes back to the *dev* file system and consumed by the file system, coordinating with devname_nslookup(), to create the resolved device name.

5 Devname Future

5.1 ClearView Vanity Network Interface Names

Currently Solaris network interface names are tied to the underlying network hardware on each machine. A network interface is named after its hardware instance number. This make it difficult to rename a network device. Tools, like dladm, are provided for sysadmins to rename a network device easier, and a /dev/net directory is dedicated to host these names.

The /dev/net names have a common characteristic, which is dynamic binding to different VLAN, aggregation or other properties. It would be very complex, almost impossible, to implement such a devfsadm link generator to handle the dynamic creation of the /dev/net names.

The DBNR mechanism provided by the *dev* file system would solve the problem cleanly. Basically, the dynamic name binding resolution for /dev/net devices can be handled inside a special lookup routine. The routine will be invoked by the file system when a lookup on the /dev/net device is intercepted. This routine can be delivered through a plug-in module that is dedicated to handle the dynamic /dev/net device name binding resolutions.

5.2 Dynamic Device Name Creation

On many occasions, new /dev names need to be created in response to certain events. For example, the /dev/lofi device creation falls into this category. In this case, the application must issue an ioctl to a driver to create new minor nodes. This creates a sysevent indicating a minor node creation. The event is transported from kernel to syseventd in userland, which notifies devfsadmd. The devfsadmd then makes a libdevinfo call to get the minor node information from kernel, finds the appropriate link generator module for the device, and creates a new name in /dev.

Potentially, the described complex approach can be simplified with the DBNR technology, just like the /dev/pts namespace.

5.3 Centralized Tape Device Naming

As we discussed earlier, devname project connects the local host devic ename resolution with the centralized name services. A framework is in place to be deployed for individual device types.

With the fact that a customer has been asking for consistent device name of the same tape drive in their enterprise, we see a good opportunity to deploy the “devname” support in a NIS environment, or even extend to support LDAP as well. Technical expertise in the tape drive area is needed to define appropriate naming scheme for tape devices. User friendly sysadmin tools to manage the name service database should also be included on the deliverable list.

5.4 Enhancements to Zones Device Support

One notable functionality missing in Zones is setting up an anonymous ftp environment. The

ideal approach is to permit Zone admin to instantiate a new filesystem instance containing a subset of devices available within the zone's /dev. This requires additional work in the filesystem layer. A simpler approach is to create /dev/ftp, with ftpconfig performing a loopback mount of /dev/ftp to ftp/dev.

It is desirable to build a new utility on top of libdevinfo for modifying the content of non-global /dev instances. This will permit zones device matching rules to be modified dynamically without a reboot.

Finally, the default permission of devices is based on owner/group id in the global zone. If name to id mapping is different between global and local zone, the default permission would be wrong in the local zone. Fortunately, existing entries in /etc/minor_perm are limited to root, sys, tty, and uucp, so it is not yet a problem.

5.5 Root Device Virtualization

Virtualized device names, such as the root disk (called /dev/root), would permit simplification of the Solaris deployment process.

The devname project builds a technology that can be deployed to create a generic root device name. Basically, the *dev* file system can intercept accesses to the root device, and associate the root device name to the actual physical root disk attached to the host. Expected technical assistances are in the platform support area.

5.6 Friendly Names for GUID-based Devices

Today, devices are named by driver developers through a link generation module. Some of the names contain long hex digits. There is no supported mechanism for the end user to modify the name or to add new names. Attempts have been made in the past to introduce a host-based table, permitting the user to map device names to "friendly" names. This works to some degree for local devices (see PSARC/1999/412 -- "Friendly Names"), but fails if the device needs to be named consistently across all hosts in a domain. The file system and name service communication channel established by this project cab be further deployed by the Friendly Names project to implement a centralized device name mapping table, thus overcome the scalability issues.

6 APPENDIX

A. DBNR Plug-In Module Implementation

As we mentioned earlier, the project supports a new type of kernel plug-in module to facilitate the file system DBNR agent development.

The first two sections describe how to implement an DBNR plug-in module, and configure it into the *dev* file system. The last section gives a skeleton source code on how to implement a module.

A.1 Plug-in Module Implementation

In a nutshell, net_mod implementation follows all the conventions in kernel module development. In this case, following specifications are defined:

- mod_devfsops is the modops structure for DBNR modules.
- struct devname_ops defines the module ops that can be implemented in a DBNR module.

```
/*
 * directory vnode ops implemented in a loadable module
 */
struct devname_ops {
        int     devnops_rev;    /* module build version */
        int     (*devnops_lookup)(char *, devname_handle_t *, struct cred *);
        int     (*devnops_remove)(devname_handle_t *);
                int     (*devnops_rename)(devname_handle_t *, char *);
        int     (*devnops_getattr)(devname_handle_t *, struct vattr *, struct cred *);
        int     (*devname_readdir)(devname_handle_t *, struct cred *);
};
```

Here, the devname_handle_t is an opaque handle representing a *dev* file system node.

- Interfaces are provided for the module writers to retrieve vnode, path-name, name, and parent information from an opaque devname_handle. Listed here are the interfaces:

```
devname_handle_t - An opaque handle for a devname filesystem node
devname_get_vnode - Retrieve the vnode from a handle
devname_get_path - Retrieve the node path from a handle
devname_get_name - Retrieve the node name from a handle
devname_get_dir_handle - Retrieve the parent directory's devname_handle
                         from a child handle
```

An example plug-in module skeleton is provided in the Appendix section to demonstrate how these interfaces may be used.

A.2 DBNR Plug-in Module Installation

Following steps are needed to install a module:

- step 1. install the module binary, e.g. subdir_mod, at /kernel/devname;

- step 2. update the /etc/dev/devname_master with following entry:

  /dev/[subdir] module=subdir_mod

  Here, the subdir_mod is the module binary name, and /dev/subdir is the subdirectory with all the device names resolved by subdir_mod.

- step 3. Run "devfsadm -m" at the end of the installation script.

  This command will pass the /etc/dev/devname_master configurations down to the *dev* file system in the kernel.This step is not needed if the system is rebooted right after module is installed because the devfsadm automatically scans the contents of /etc/dev/devname_master at system reboot.

A.3 Skeleton Source Code for a DBNR Plug-in Module

```
#include <sys/fs/sdev_impl.h>
#include <sys/fs/sdev_node.h>

static int devname_lookup(char *, devname_handle_t *, struct cred *);
static int devname_remove(devname_handle_t *);
static int devname_rename(devname_handle_t *, char *);
static int devname_readdir(devname_handle_t *, struct cred *);
static int devname_getattr(devname_handle_t *, struct vattr *,  struct cred *);

static struct devname_ops devname_ops = {
        DEVNOPS_REV,            /* devnops_rev, */
        devname_lookup,         /* devnops_lookup */
        devname_remove,         /* devnops_remove */
        devname_rename,         /* devnops_rename */
        devname_getattr,        /* devnops_getattr */
        devname_readdir         /* devname_readdir */
};

/*
 * Module linkage information
 */
static struct modldev modldev = {
        &mod_devfsops,
        "devname DBNR mod 1.0",
        &devname_ops,
};
static struct modlinkage modlinkage = {
```

```
	MODREV_1, &modldev, NULL
};

int
_init(void)
{
	return (mod_install(&modlinkage));
}

int
_fini(void)
{
	return (mod_remove(&modlinkage));
}

int
_info(struct modinfo *modinfop)
{
	return (mod_info(&modlinkage, modinfop));
}

/*ARGSUSED2*/
static int
devname_lookup(char *nm, devname_handle_t *dhl, struct cred *cred)
{
	int error = 0;
	char *dir = NULL;
	char *link = "/devices/pseudo/mm@0:zero";

	error = devname_get_dir_path(dhl, &dir);
	if (error) {
		error = ENOENT;
		goto errout;
	}

	/*
	 * returns the physical path for /dev/pseudo/zero
	 */
	if (strcmp(dir, "/dev/pseudo") == 0) && strcmp(nm, "zero") == 0)
		link = "/devices/pseudo/mm@0:zero";

	devname_set_nodetype(dhl, (void *)link, (int)DEVNAME_NS_PATH);

errout:
	return(error);
}

/*ARGSUSED*/
```

```
static int
devname_readdir(devname_handle_t *hdl, struct cred *cred)
{
        int error = 0;
        char *entry;
        char *dir;

        (void) devname_get_name(hdl, &entry);
        (void) devname_get_dir_name(hdl, &dir);

        /*
         * could do validity check for the entry
         * or do book keeping
         */
        return (0);
}

/*ARGSUSED*/
static int
devname_remove(devname_handle_t *hdl)
{
        char *entry;

        (void) devname_get_name(hdl, &entry);

        /*
         * could sanity checking on the entry
         * and disallow delete
         */

        return (EROFS);
}

/*ARGSUSED*/
static int
devname_rename(devname_handle_t *ohdl, char *new_name)
{
        char *oname;

        (void) devname_get_name(ohdl, &oname);

        /*
         * make sure the oname can be renamed  to new_name
         * reject the operation if not
         */

        return (ENOTSUP);
}
```

```
/*ARGSUSED*/
static int
devname_getattr(devname_handle_t *hdl, vattr_t *vap, struct cred *cred)
{
        struct vnode *vp;

        error = devname_get_vnode(hdl, &vp);
        if (error)
                return (error);

        /*
         * fill up the vap structure or
         * leave it to the filesystem or
         * check the vnode and do sanity checking
         */

        return (0);
}
```

B. Local Instance Profiles and Interfaces

B.1 How to Mount and Export /dev Devices into Local *dev* Instance

To create a new instance at /zone/dev, simply mount the filesystem when a local zone is booting:

```
/usr/sbin/mount -F dev dummy_arg /zone/dev
```

To make device entries appear, local zone configuration, i.e. Zonecfg(1M), call a set of libdevinfo interfaces and pass down a set of matching rules to be applied against the global instance of /dev name space. The following code (omitting error checking) adds null, zero, and tape drives to /zone/dev.

```
di_prof_t myprofile;
di_prof_init("/zone/dev", &myprofile);
di_prof_add_dev(myprofile, "null");
di_prof_add_dev(myprofile, "zero");
di_prof_add_dev(myprofile, "rmt/*");
di_prof_commit(myprofile);
di_prof_fini(myprofile);
```

To remove a device (e.g. c0t0d0), issue the following calls:

```
di_prof_init("/zone/dev", &myprofile);
di_prof_add_exclude(myprofile, "dsk/c0t0d0*");
di_prof_add_exclude(myprofile, "rdsk/c0t0d0*");
di_prof_commit(myprofile);
di_prof_fini(myprofile).
```

B.2 Local Instance Profile

A dev filesystem's profile is a set of matching rules that specify what entries are accessible on that filesystem. There are four types of entries in a profile: include list, exclude list, symlinks, and name mappings.

o Include List

An include list is a list of device files and directories under the global /dev which are included in the non-global filesystem, e.g.:

```
cpu/self/cpuid
cua/*
zero
[r]dsk/c0*
rmt
bogus/path/*
```

In the above example, the device files /dev/cpu/self/cpuid, /dev/cua/* and /dev/zero are accessible from the non-global filesystem. The [r]dsk/c0* makes all disks under c0 visible, both raw and cooked entries. Since /dev/rmt is a directory, a subdirectory named "rmt" is created in the non-global filesystem, but the direct has no contents. Finally, the entry bogus/path/* above is ignored since there is no corresponding /dev/bogus/path/* in the global /dev.

o Exclude List

An exclude list is the opposite of include list. It is primarily used as a way to dynamically remove a device from a Zone. For exmaple, a removable media /dev/cdrom may be dynamically assigned to a zone via

```
include cdrom -- di_prof_add_dev(prof, "cdrom");
```

and later deassigned with

```
exclude cdrom          -- di_prof_add_exclude(prof, "cdrom");
```

The exclude list can also be used together with include patterns to express a list of devices that are difficult to express with include alone. For example, to export all disks except root (c0d0s0) and swap slices (c0d0s1)

```
include [r[dsk/*
exclude [r]dsk/c0d0s[0-1]
```

o Symlinks

A profile may also include a list of symbolic links which will be created in the non-global dev filesystem. Symbolic links are strictly within the non-global instance and do not reference the global /dev. Symbolic links are specified as:

```
stdin -> fd/0
stdout -> fd/1
stderr -> fd/2
```

No globbing patterns are allowed in symlinks. Any globbing metacharacter is treated as a regular character.

o Name Mappings

Name mapping list is similar to the include except the the device is referenced by a different name from the global /dev. Each entry in the mapping list contains two names, one is the global /dev and the other the local /dev name.

```
conslog = ftp/conslog
null = ftp/null
```

The above mapping extries map /dev/conslog and /dev/null in to local /dev as /dev/ftp/conslog and /dev/ftp/null, respectively. No globbing patterns are allowed in name

mapping specifications.

Note that it is possible to specific matching rules resulting in local name conflicts. For example:

```
include console
zones/zone1/zoneconsole = console
```

The first rule specifies local /dev/console to be the global /dev/console. The second rule maps zone console in global /dev/zones/zone1/zoneconsole to local /dev/console. It is the responsibility of the consumer to ensure that such conflicts do not happen.

B.3 Libdevinfo APIs for Local Profile

The following routines in libdevinfo are used to construct matching rules discussed above and apply them to a non-global dev filesystem.

int di_prof_init(const char *mountpt, di_prof_t *profp)

Allocate an opaque handle to dev filesystem mounted at the mount point "mountpt". The data structure is returned in the area pointed to by "profp".

void di_prof_fini(di_prof_t prof)

Free up the data structure allocated by a previous call to di_prof_init().

int di_prof_commit(di_prof_t prof)

Commits the profile to the dev filesystem.

int di_prof_add_dev(di_prof_t prof, const char *dev)

Add a directory or device file "dev" to the profile's include list.

int di_prof_add_exclude(di_prof_t prof, const char *dev)

Add a directory or device file "dev" to the profile's exclude list.

int di_prof_add_symlink(di_prof_t prof, const char *linkname, const char *target)

Add a symbolic link, specified by "linkname" and "target", to the profile's symlink list.

int di_prof_add_map(di_prof_t prof, const char *source, const char *target)

Adds a name mapping entry, specified by "source" (the global /dev name) and "target" (the local /dev name), to the profile's name mapping list.

For example, to add following profile:

Include list:
fd
dsk/*
zero

Exclude list:
dsk/c[2-4]*
dsk/xyz

Symlink list:
stdin -> fd/0
stdout -> fd/1
stderr -> fd/2

Name mapping list:
conslog = ftp/conslog
null = ftp/null
null = mydir/mynull

the following calls can be made:

```
di_prof_init("/my/dev/mountpt", &myprofile);

di_prof_add_dev(myprofile, "fd");
di_prof_add_dev(myprofile, "dsk/*");
di_prof_add_dev(myprofile, "zero");

di_prof_add_exclude(myprofile, "dsk/c[2-4]*");
di_prof_add_exclude(myprofile, "dsk/xyz");

di_prof_add_symlink(myprofile, "stdin", "fd/0");
di_prof_add_symlink(myprofile, "stdout", "fd/1");
di_prof_add_symlink(myprofile, "stderr", "fd/2");

di_prof_add_map(myprofile, "conslog", "ftp/conslog);
di_prof_add_map(myprofile, "null", "ftp/null);
di_prof_add_map(myprofile, "null", "mydir/mynull);

di_prof_commit(myprofile);
di_prof_fini(myprofile).
```

C. **Libdevinfo APIs for Backing Store Access**

NAME

finddev - search /dev without triggering reconfiguration

SYNOPSIS

```
int device_exists(const char *devname)
int finddev_readdir(const char *dir, finddevhdl_t *handlep)
int finddev_by_name(const char *expr, finddevhdl_t *handlep)
void finddev_close(finddevhdl_t handle)
const char *finddev_next(finddevhdl_t handle)

#include <libdevinfo.h>
```

PARAMETERS

devname	pathname of a /dev device
dir	pathname of a /dev directory
handlep	address of a pointer to a finddevhdl_t
expr	pathname of a /dev regex expression
handle	pointer to a finddevhdl_t

DESCRIPTION

device_exists() returns an indication whether the named device exists. The path must be a path in /dev.

finddev_readdir() allocates a finddevhdl_t, performs a readdir on the named directory, and associates the resulting file list with the handle. The allocated handle is returned in *handlep on success. The path must be a path in /dev.

finddev_by_name() allocates a finddevhdl_t, performs a readdir on the named directory, and filters the filelist through the expression expr. The allocated handle is returned in *handlep on success. The path must be a path in /dev.

finddev_close() frees a handle returned by a successful call to finddev_readdir() or finddev_by_name().

finddev_next() returns a pointer to the next path in the filelist referenced by the handle. For the first call for a handle, it returns the first path in the list.

RETURN VALUES

device_exists() returns 1 if the devname exists and 0 otherwise.

finddev_readdir() returns 0 or an errno if an error occurred.

finddev_by_name() return 0 or an errno if an error occurred.

finddev_next() returns a pointer to a string or NULL.

ERRORS

EINVAL - illegal value
ENOENT - file not found
ENOTDIR - not a directory

EXAMPLES

Here is an example of reading a /dev directory:

```
finddevhdl_t handle;

if (finddev_readdir(dir, &handle) == 0) {
        while ((name = finddev_next(handle)) != NULL) {
                /* process filename */
        }
        finddev_close(handle);
}
```

D. An Example “devname” Deployment in NIS Environment

Note, this project does not provide this feature to external Solaris end user yet.

Assuming someone has a tape drive. This tape drive has a WWN as "200000012B37C2D45200061". Following are the steps to name the tape drive as "/dev/tape/HR-backup", across the enterprise which runs NIS name service:

step 1, modify the /etc/nsswitch.conf by adding the entry of

devname nis

step 2, modify the /etc/dev/devname_master by adding the entry of

#/dev directory directory map name

/dev/tape tapes

step 3, create the /etc/dev/tapes map file with following entry

#/dev/tape/device_name device property

HR-backup devices_wwn="WWN200000012B37C2D45200061"

step 4, translate both /etc/dev/devname_master and /etc/dev/tapes into NIS database files and deposit the database into where the other NIS maps resides. In order to do that, first, add the "/etc/dev/devname_master" and "/etc/dev/tapes" entries into the NIS service makefile, /var/yp/Makefile on the master NIS server. Then run the makefile and produce the NIS database formatted file for devname_master and tapes. Note, this step is normally performed by the NIS service administrators. What they normally need is the map name and map contents in an ASCII text file.

step 5, replacing the contents in the /etc/dev/devname_master file with following entry

+devname_master

note, the "+" indicates the map named "devname_master" is on a name server designated for this host. same for next step.

step 6, replacing the contents in /etc/dev/tapes with this entry

+tapes

step 7, Final step is to push down NIS client configuration changes to the software programs, i.e. the name service switch and devname filesystem. This is what is needed:

- restart the "/usr/lib/netsvc/yp/ypbind" service;

- run "devfsadm -m".

Note, step 5, 6, and 7 can be repeated on any other NIS clients that are running the devname bits.

After these operations, access to the "/dev/tape/HR-backup" tape device is going to be directed to the same tape drive on the hosts configured with the same NIS server as mentioned in step 4.

Another note is, local map entries can be added into /etc/dev/devname_master and /etc/dev/tapes. The only difference is that maps in these local files are not visible outside of the local hosts. It is probably not too interesting either since we are talking about centralized device naming here.