Solaris Hotplug Framework: Architecture and Design

Revision 1.8

July 21, 2009

Colin.Zou@Sun.COM

Govinda.Tatti@Sun.COM

Zhijun.Fu@Sun.COM

Evan.Yan@Sun.COM

SHP Project Team

Sun Microsystems, Inc.

Table of Contents

1. Introduction

1.1 Background & Motivation

In Solaris today, every hotplug capable I/O bus technology (e.g., PCI, USB, PCMCIA, Storage) has its own hotplug framework. There is no generic hotplug framework in Solaris kernel. Therefore, each hotplug capable I/O bus needs to invent its own interfaces and mechanism.

In the userland, although there is a popular management software cfgadm, it also requires that each bus to have a bus specific plugin.

Therefore, either in Solaris kernel or in userland, there are rooms to improve the current software architecture by having a generic hotplug framework.

If look at the requirements, first, more and more I/O buses are supporting hotplug. That is, we need to develop new bus specific hotplug frameworks from time to time if without a generic Solaris hotplug framework there. And also, we need to maintain more and more bus specific hotplug frameworks as time going on.

Second, the hotplug requirement under virtualized environments (e.g., I/O Virtualization in Ldoms or Xen) raised a new issue here: arbitrary devices in the device tree should be able to be "hotplugged" in order to migrate between domains. This is a significant requirement and one of the major motivations of this project. See section 1.2 and section 2.5.5 for more details.

1.2 Goal

The main goal of this project is to deliver a generic common hotplug framework to Solaris users, a foundation which can support the hotplug functionality for any hotpluggable bus and also, to support device migration functionality in virtualized environment through "virtual hotplug" (see section 1.3 for this term). This new Solaris hotplug framework has a state machine based architecture. It includes generic hotplug states and provides various common hotplug interfaces, which includes:

- Management Interfaces

  Generic or bus specific hotplug administration applications including GUIs can be written using these interfaces.

- Hotplug Event Interfaces

  Hotplug-aware consumers such as drivers, kernel modules and hotplug aware applications can receive both synchronous and asynchronous hotplug events.

- Device Driver Interfaces

  Device drivers can work with the hotplug framework to participate in hotplug activities such as surprise removal, hot replacement, dynamic resource balancing, error handling.

- Hotplug Controller Driver (HPC) Interfaces

  HPC drivers will communicate with the hotplug framework through these interfaces. New HPC driver can be written easily for any hotpluggable bus.

This document proposes a new generic Solaris hotplug framework that lays the foundation to address current and future hotplug requirements.

1.3 Terminologies

Physical Hotplug	Hotplug operations on the hardware receptacle.
Virtual Hotplug	Hotplug operations at an arbitrary device node in the Solaris device tree.
Connector	Place where physical hotplug happens.
Port	Place where virtual hotplug happens.
Connection	Place where hotplug (either physical or virtual) happens.
Component	Physical object being hotplugged.
Device	Software object being hotplugged.

Hotplug-object Object being hotplugged (either physical or virtual).

1.4 Scope and Road Map

Currently, there are a lot of issues need to be addressed in hotplug domain. It should be a phase by phase effort to carry out all of them. This case is the first phase and it focuses on the following:

Phase-I:

- Basic hotplug framework features and interfaces
- Virtual hotplug support for PCI/PCIe devices
- Physical hotplug support for PCIe (Native, ACPI) and PCI SHPC (Standard Hot-Plug Controller, see [2] in section 1.5) devices in new hotplug framework
- Hotplug userland stack (daemon, library and CLI)
- A new cfgadm plugin to support cfgadm(1M) to work with the new hotplug framework

This document focuses on the overall architecture and also interfaces exported by kernel modules. Other documents in this case give the hotplug userland stack architecture details and interfaces.

Besides the framework implementation, this phase also ports PCI/PCIe hotplug controller drivers to the new hotplug framework. Any other driver porting jobs are out of scope of this case.

The following are what can be foreseen for the future phases but out of scope of this case:

Phase-II:

- Hotplug events through Device Contract and LDI frameworks

  Device contract and LDI frameworks enhancements to support new hotplug events.

- New hotplug DDI interfaces for device drivers

  This task develops DDI interfaces for interaction between hotplug framework and device drivers.

- New PCI resource allocator and a common PCI Configurator with support for dynamic resource re-balancing

  This task is to minimum the chance that a PCI hotplug operation would fail due to lacking of resource (e.g., bus numbers, I/O or memory address spaces, etc.) at connector.

- Support hotplug FMA for PCIe devices

Phase-III:

- Support for GUI administer tool
- Support for ExpressCard

  ExpressCard is a industry standard replacing PCMCIA cards. This task is to develop a new hotplug controller driver based on the new hotplug framework interfaces.

In short, this case focuses on the infrastructures, and future phases will add more features based on what are being done here.

1.5 References

[1] PCI Hot-Plug Specification, Revision 1.0, June 20, 2001

[2] PCI Standard Hot-Plug Controller Specification, Revision 1.0, June 20, 2001

[3] PCI Express Base Specification, Revision 2.0, Dec 20, 2006

[4] Solaris Operating System - Hardware Virtualization Product Architecture, Revision 1.0, November 2007

[5] Hotplug related ARC cases:

PSARC/1993/687	Hot Plugging
PSARC/1996/285	DR for CPU/Memory Boards

PSARC/1998/327	PCI Hotplug Support
PSARC/1998/460	RCM Framework
PSARC/1999/122	PCI Bus Resource Allocator
PSARC/1999/135	Solaris 8 RAS Enhancements
PSARC/1999/287	Solaris Embedded Fcode Interpreter
PSARC/1999/322	System Event Framework
PSARC/1999/686	Promote PCI Hotplug Interfaces
PSARC/2000/085	Extensions for PCI Hotplug Framework
PSARC/2000/218	RCM Framework Enhancements
PSARC/2000/295	PICL Event Interfaces
PSARC/2000/298	PCI Hotplug slot registration policy
PSARC/2000/532	State Model for Device Information Nodes
PSARC/2002/315	cPCI Autoconfiguration Support
PSARC/2003/193	Solaris Contracts and restart agreements
FWARC/2005/048	1275 Bindings for PCI Express Interconnect
PSARC/2005/375	PCI Hotplug Extensions for PCIe
PSARC/2006/037	PCI Express Hotplug Framework Interrupt Interfaces
FWARC/2006/198	PCI Hotplug Resource Preallocation
PSARC/2007/197	ZFS Hotplug
PSARC/2007/290	Retire Agent for IO devices

2. Architecture Overview

2.1 Physical Hotplug & Virtual Hotplug

2.1.1 Concepts

When a hardware device is plugged to system during running time, in order to configure the newly added device, operating system needs to probe the device, allocate dev_info node and resource (e.g. IO/MEM address spaces) for it, and then tries to attach driver. It is a reversed case when a hardware device is unplugged from system. This is the physical hotplug expected to be supported by an operating system.

If not limit hotplug operations to physical hotpluggable devices, that is, apply the similar operations for an arbitrary dev_info node in the device tree, then this is referred as “virtual hotplug”. For example, hot add/remove an on-board PCI NIC device. Although the device is not physically plug/unplugged to/from the system, from the software view, it is coming to the device tree when virtual plugged and going when virtual unplugged.

2.1.2 Background & Goal

Currently, the hotplug operations supported on Solaris are limited to those devices associated with a hotpluggable physical receptacle (mentioned as “connector” in this document). This is almost fine in the single domain environment in the past, but, nowadays, more and more customers are adopting virtualized environment. Therefore, a requirement is raised that arbitrary device nodes in the device tree should be able to be “hotplugged” and then migrated between domains.

This case tries to provides a uniformed and consistent hotplug model that is capable of adding and removing arbitrary devices with or without hotpluggable physical receptacles being present, that is, support both physical hotplug and virtual hotplug within one framework.

2.2 Overall Architecture

This section gives an overview of the new Solaris Hotplug Framework. Figure 1 shows the block diagram of software components involved.

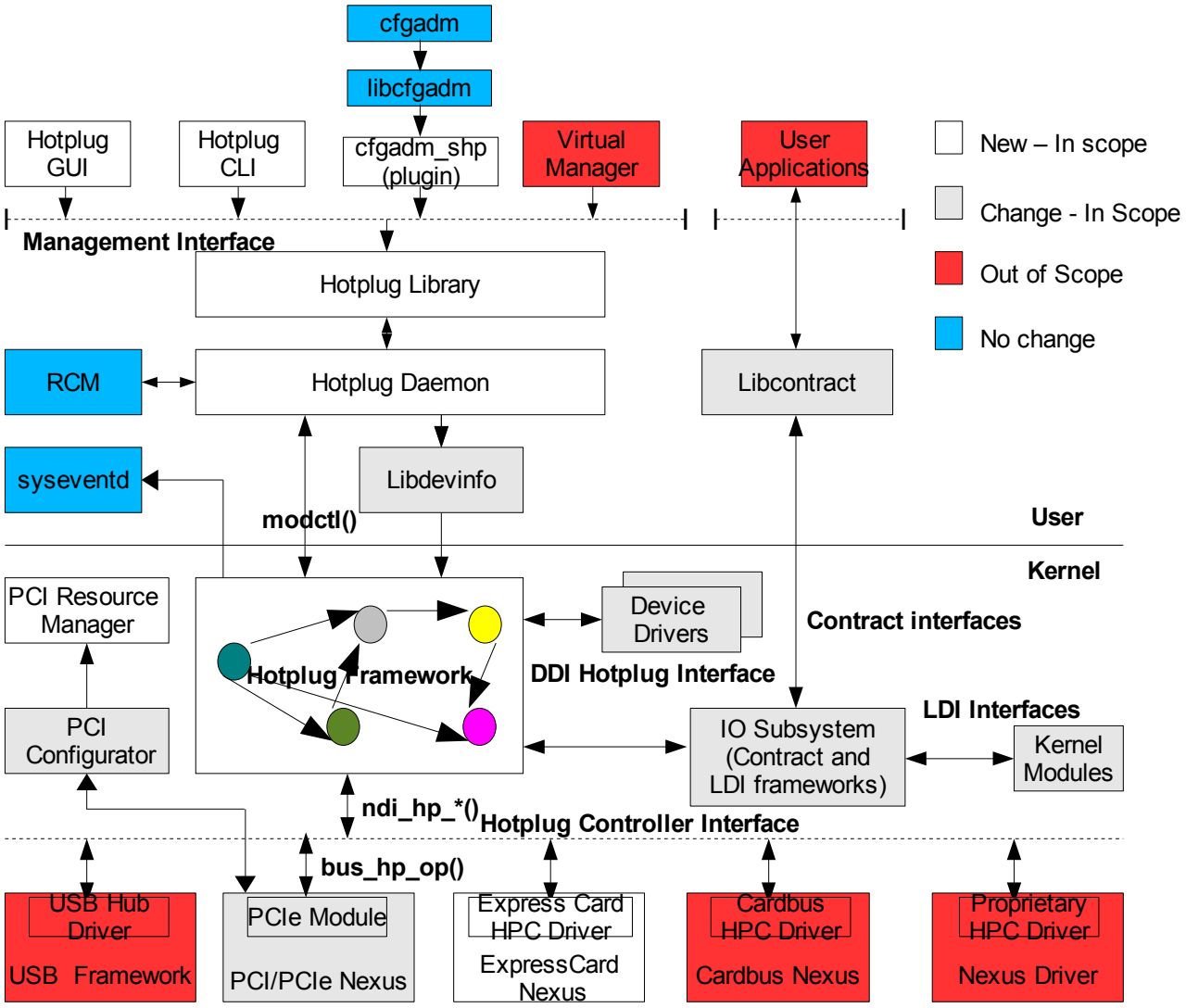

Figure 1. Overall Architecture of Solaris Hotplug Framework

The new Solaris hotplug framework is based on a state machine which is supposed to include generic hotplug states and transitions between states. Refer to section 2.5 for details.

It provides a set of generic and common hotplug interfaces, which includes:

- Management Interfaces

  Generic or bus specific hotplug admin applications including GUIs, can be developed using the newly defined management interfaces provided by the hotplug library. Refer to userland design documents of this case for details.

- Hotplug Controller Driver Interfaces

  Hotplug controller drivers can be developed for any hotpluggable bus using the newly defined bus_op and NDI interfaces (bus_hp_op() and ndi_hp_*() in Figure 1). Refer to section 3.3 for details.

For the kernel – userland interfaces, a new command is added to the existing modctl system call and some new interfaces

are added to libdevinfo. See section 3.1 and 3.2 for details.

From the long term view, cfgadm users are supposed to switch to the hotplug CLI provided by this case. Before that, to make cfgadm running on this framework, so that hotplug controller drivers don't need to maintain cfgadm ioctls any more, a new cfgadm plugin will be developed as the translator from cfgadm commands to Solaris hotplug framework commands. Refer to section 2.8 for details.

And there are plans for some other interfaces to be developed in future phases, such as new hotplug DDI interfaces for device drivers, hotplug events through Device Contract and LDI frameworks, etc. See section 1.4 for details.

2.3 Connector & Port

In this document, either “connector” or “port” is a software object standing for the place where hotplug operations happen.

A connector is corresponding to a hardware receptacle, e.g., a PCI slot, USB port, cardbus slot, SAS port, and fiber channel port, etc. The occupant of the connector is mentioned as “component” in this document. It is corresponding to a PCI card, USB device, cardbus card, etc.

A port is used to specify a “place” under a nexus dev_info node in the device tree in a running Solaris image. For example, “/pci@0,0 pci.1,2” specifies a place under a PCI nexus dev_info node /pci@0,0, and at that place, a dev_info node with attributes “PCI device number 1 and function number 2” is expected. For details of the naming of a connector or port, refer to section 2.4. The occupant of the port is mentioned as “device” in this document. It is corresponding to a dev_info node in device tree. Note, it is forced that the relationship between port and device is 1 : 1. By default, each initialized dev_info node will have a port if the node's bus software (e.g., if PCI dev_info node, its bus software is PCI nexus driver) supports virtual hotplug.

The same data structure is defined for both connector and port. The data structures of the connectors and ports with the same parent dev_info node are put to a link which is linked to the parent dev_info node, so that each connector or port is associated with a parent node in the device tree. And, the hotplug states of connector or port are unified in one state machine. Thus, Solaris hotplug framework provides consistent interfaces for the hotplug operations happened at either connector or port.

2.4 Naming of Connection

A “connection” is either a connector or a port. Any connections in a system are specified in this way:

> “*Absolute path to parent dev_info node*” + “*connection name*”

A *connection name* is a string which is unique among all of the connections which are immediately under the same parent dev_info node. So that the above “path + name” combination is system wide unique.

A connection's name is decided by bus software (hotplug controller driver). For connector, it is usually derived from hardware properties, e.g., the hardware slot name set by platform manufacturer. For port, it is usually set according to bus specific industry standard.

An example is:

/pci@0,0/pci10de,5d@e NEM0

“NEM0” is the PCI slot name get from “slot-names” property which is from platform manufacturer.

For virtual hotplug case, the port name is set by hotplug controller drivers (instead of Solaris hotplug framework). Because only the bus specific software has the knowledge of naming a receptacle under the bus node.

An example is:

/pci@0,0/pci10de,5d@e pci.1,2

“pci.1,2” is the port name which stands for PCI device number 1 and function number 2. It specifies a device place under a PCI bridge node according to PCI industry standard.

Solaris hotplug framework does not define any specific naming rules for connection names, but it assumes that a hotplug controller driver instance will give names which are unique among the immediate children of the dev_info node that the driver instance is being attached.

2.5 Hotplug States & State Machine

2.5.1 Overview

Solaris hotplug framework defines a set of hotplug states and a hotplug state machine, see Figure 2. Here, “hotplug states” are the states of a connection or a hotplug-object. They are exposed to both the hotplug controller drivers and hotplug userland stack. Thus, the hotplug userland stack, framework and controller drivers can then work on the same states and state machine.

The hotplug states are supposed to be generic, that is, it is not aimed to enumerate every possible states in various hotpluggable buses. Usually, the bus specific states are much more detail than the hotplug states defined by the framework. Hotplug controller drivers can run on their own states but they should aware the framework states and have mappings between them when interacting with the framework.

2.5.2 Connector States

To support both physical hotplug and virtual hotplug in one state machine, states for connector, port and device are defined as three sub-sets of states. And we can see the obvious dependencies between them in the state machine.

The first sub-set of states are for connectors:

- **Empty**: No component plugged to the connector.
- **Present**: Component already plugged to the connector.
- **Powered**: The connector is powered.
- **Enabled**: The connector is fully functional.

As mentioned in section 2.3, a connector specifies a physical hotpluggable place. For example, a PCI hotpluggable slot. The following is the example steps of the connector state transition:

1. User plugs a PCI board to an empty slot when system is running.
2. The PCI hotplug controller driver detects there is something plugged in the slot, then, the corresponding connector’s state is turned to “Present” from “Empty”.
3. User issues a command to power the slot. The PCI hotplug controller driver gets the command from hotplug framework and then programs the slot hardware registers to add power to the slot. When this is done, the connector’s state is turned to “Powered”.
4. There might be some more detail programming jobs need to be done for the connector before the plugged PCI board can be accessed by software. User issues another command to “enable” the connector. The connector state is then turned to “Enabled”. Now, the newly plugged PCI board is ready to be probed, that is, OS software can then read the configure space of it and knows what kind of device it is.

Note, because the connectors are managed by nexus drivers (PCI hotplug controller driver in the above example), the power management related issues are handled by the nexus drivers and system PM modules. Hotplug controller drivers should request hotplug framework to update the connector’s state whenever it is changed for some reasons. On the other way, hotplug framework passes the state change commands (e.g., power the connector) to the hotplug controller driver and the latter take care of the rest jobs including PM operations.

The above states and state transitions are what must be experienced before a hotplugged hardware device could be probed by Solaris software. This process is the “physical hotplug” mentioned in section 2.1.

2.5.3 Port and Device States

After the physical device is probed, some dev_info nodes will be allocated for the newly plugged device. The process of linking the dev_info nodes to the right place in the device tree is taken as “virtual hotplug” mentioned in section 2.1.

Let’s continue with the above example. Suppose the newly plugged PCI board has four functions, we know that the existing Solaris PCI configuration software will probe the device's functions one by one by reading their configuration registers, and then, it allocates a dev_info node for each function and link the four new nodes as children nodes to the PCI nexus node who manages the PCI slot.

According to PCI industry standard, one PCI device (either hotplugged device or on-board device) takes one device number and one or more PCI function numbers. By specifying the PCI nexus node and a device/function number pair, we can

specify a unique “place” in a PCI device tree. Therefore, we know that we can specify a “port” (refer to section 2.3) for a PCI dev_info node by specifying its device number, function number and also the absolute path to its parent node.

It is straightforward that the port has the following two states:

- **Port-Empty**: the port has no device occupying it.
- **Port-Present**: the port is occupied by a device.

One port is for one dev_info node only. In the above example, four ports will be created and the four new nodes will be virtually “plugged” to their ports one by one.

Note, if the bus software supports virtual hotplug functionality, then, for all the nexus nodes of this bus, ports will be created for each immediate child of the nexus node when the child node is initialized, no matter the device is a physical hotplugged device or an on-board device. Thus, arbitrary dev_info nodes can be virtually hotplugged.

When the port state goes to “Port-Present”, user can then manipulate the port's device state. It can be one of the following state:

- **Initialize**: The dev_info node is created and not get to “Probed” state yet.
- **Probed**: The driver's probe(9e) routine has been invoked and it returned success.
- **Attached**: The driver's attach(9e) routine has been invoked and it returned success.
- **Operational**: Post attach processing, including DACF, has been completed, and the device is fully functional.
- **Maintenance**: The device is partly functional or not functional at all. This is either because of a fault or a result of an intentionally administration action. It could has some sub-states, but the definition of the sub-states are out of scope of this case (planned in future phases).

There is a straightforward mapping between the above device states and the DS_* states proposed by PSARC/2000/532.

Table 1. State Mapping

devi_node_state	Port's Device State
DS_PROTO	Initialize
DS_LINKED	Initialize
DS_BOUND	Initialize
DS_INITIALIZED	Initialize
DS_PROBED	Probed
DS_ATTACHED	Attached
DS_READY	Operational / Maintenance

Because the relationship between port and device is 1:1, therefore, to make it easier to tell, this document also mentions the port's device state as port state.

Continue with the above example, corresponding to the four functions in the PCI board, four new dev_info nodes are now in four ports. User can manipulate each of the port's state separately. Note, it is out of the scope of this case to support user commands to bring a port to “Maintenance” state, but it is planned in future phases.

2.5.4 State Machine

The following is the state machine.

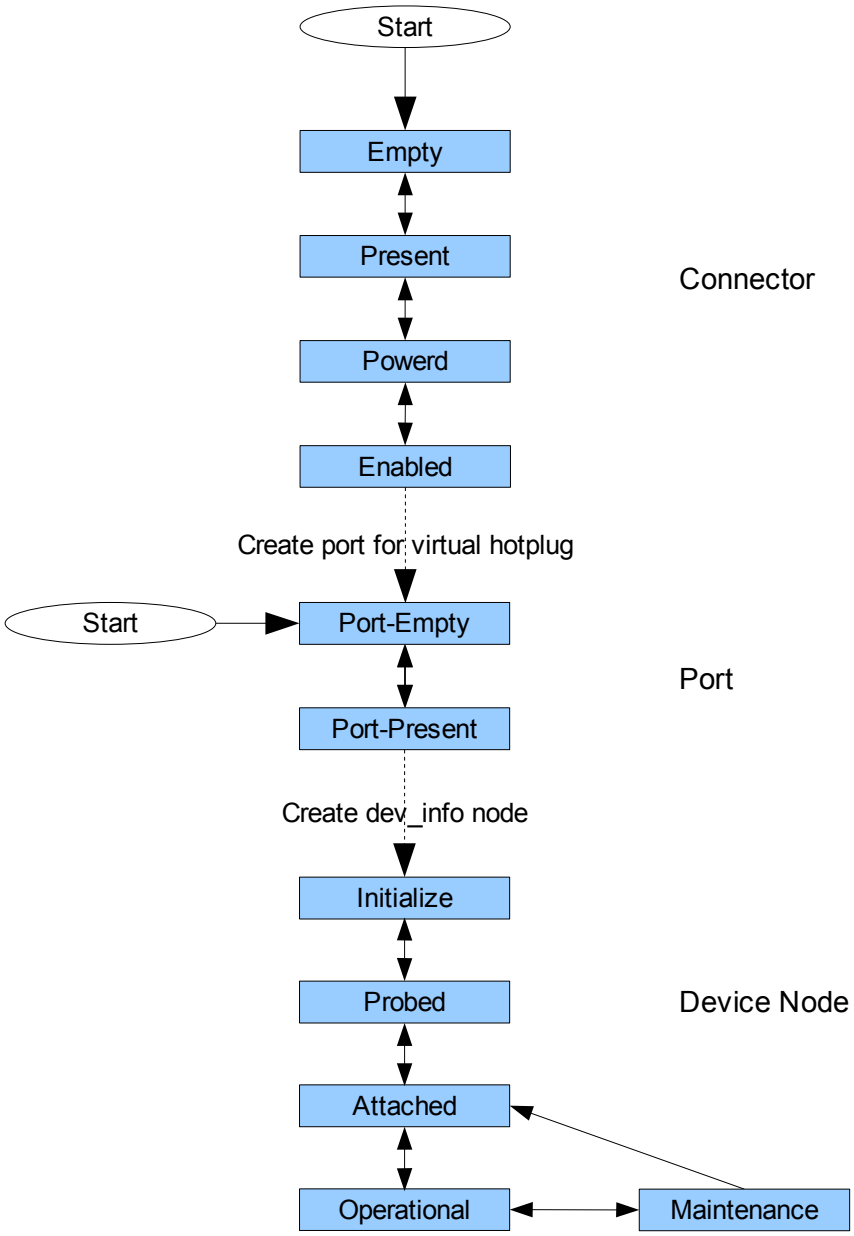

Figure 2. Hotplug State Machine

There are two start points in the state machine. The top one is for the case that a connector is involved. The other one is for the case that no connectors involved, i.e., the on-board device case.

When a connector is created, it starts from “Empty” state. For example, the PCI hotplug slot case or USB port case. The connector is created usually when hotplug controller driver discovers a physical hotpluggable receptacle in the system.

When a port is created, it starts from “Port-Empty” state. It is created for receiving a dev_info node. The port can either depend or not depend on a connector. That is, either a physical hotpluggable device or an on-board device can have a port. For a physical hotpluggable device, only if its connector is at “Enabled” state, the corresponding port and device node would exist.

The port is created mainly in two scenarios:

- When a dev_info node is being initialized by bus software (nexus driver), for example, during device enumeration

when boot or at the end of a physical hotplug process. In this case, the port state will be updated to “Port-Present” from “Port-Empty” immediately after the port is created.

- When a userland command is received to create a specific port. The port state will be kept as “Port-Empty” until a virtual hotplug operation happens.

When it is known that a port's device exist, the port's state goes to “Port-Present”. After the dev_info node of the device is allocated, the port state goes through “Initialize”, “Probed”, “Attached” and “Operational”. Now the device is fully functional. And then, if something happened (except driver detach or predetach) which made the device not fully functional, then it goes to “Maintenance” state.

2.5.5 A Usage Example in Virtualized Environment

As mentioned in section 2.1.2, one goal of this case is to support device migration in virtualized environment. The following is a usage example to elaborate how the virtual hotplug functionality can be used to do a device migration between domains.

Suppose there is a PCI NIC device in a system, and it has two functions. Each function has a corresponding network node in system device tree. The system administrator want to move the second function from root domain to an I/O domain so that the I/O domain can have direct access to that NIC function. The following are the example steps to perform this action:

1. Login the root domain. Administrator issues a command to list the ports in the system and find the port name corresponding to the second function of the PCI NIC device.
2. By specifying the port name and the absolute path to the parent node, administrator issues a command to turn the port state to “Initialize” in root domain.
3. Login the target I/O domain. By specifying the same port name and path, administrator issues a command to create the port in I/O domain. The port is in “Port-Empty” state.
4. Still in the I/O domain, administrator issues another command to the port to turn the port state to “Port-Present”. This will cause the hotplug framework and the bus software in I/O domain to do a read-only probe. The NIC function will then be discovered.
5. In the I/O domain, administrator issues the last command to the port to bring the port's device state to “Operational”. Done.

Note, the above example steps are only for the purpose of elaborating the possible details of what will happen during a device migration. The actual steps can be much simpler for a system administrator. For example, the hotplug administer tool can combine step 3, 4 and 5 into one command which includes creating port and upgrading port state to “Operational” from “Port-Empty”. The actual implementation of device migration is out of scope of this project.

2.6 Bus Specific Properties Support

Besides the generic hotplug state machine and commands, Solaris hotplug framework also provides a way to support setting/getting bus specific properties related with hotplug operations, e.g., set the LEDs of a PCI hotplug slot to on/off state or get the current states of the LEDs. The idea is, in the case of setting a property, the userland stack and Solaris hotplug framework just pack and pass name-value pairs of the property from userland to the hotplug controller driver, and then the driver does all the unpack and interpretation jobs and then set the property. In the case of getting a property, the driver packs the property and its value to name-value pairs, pass them to hotplug framework and userland stack, and then, the userland tool unpack the name-value pairs and print the output for users. Thus, to support a new bus specific hotplug property, the only things to do are to update the hotplug controller driver and also update the userland hotplug CLI man page for the new property support. There is no need to update the code of userland stack and Solaris hotplug framework.

More details for the related interfaces see section 3.1.

2.7 sysevents

From the view of the consumers of hotplug sysevents, nothing is changed by this case. Specifically, the following two sysevents are mentioned here:

- ESC_DR_AP_STATE_CHANGE
- ESC_DR_REQ

The hotplug controller drivers continue to generate ESC_DR_REQ. For example, when a PCI attention button is pressed, the PCI hotplug controller driver gets the interrupt and then generate an ESC_DR_REQ event as before, and the existing

userland consumers will also just work as before.

For ESC_DR_AP_STATE_CHANGE, there are two cases:

- Userland issues a state change request to hotplug framework, then the framework will generate this sysevent when the state is changed. For example, a user issues a command to enable a connector (that is, change the connector state to “Enabled”) via hotplug administration tool.
- The hotplug controller driver submit a state change request to hotplug framework by calling ndi_hp_state_change_req() (see section 3.3.2 for this interface), then the framework will generate this sysevent after it completes the state change job. For example, when a PCI power fault interrupt is received by the PCI hotplug controller driver's interrupt handler, the handler will submit a state change request to the framework that the PCI slot's state should be changed to a state less than “Powered” (that is, power off the slot); then, the framework will generate an ESC_DR_AP_STATE_CHANGE sysevent when the state change job is done.

Although the hotplug controller drivers are not generating ESC_DR_AP_STATE_CHANGE directly as before, the sysevent consumers does not see any changes because they just get the same events as before.

In the future phases, it is possible that we would replace the existing events and also modify the events consumers in order to support new functionality.

2.8 cfgadm shp Plugin

This is a new plugin developed to make cfgadm_pci(1M) continue to work for PCI SHPC and PCI Express hotplug as before. See Figure 1. The new cfgadm plugin works as the translator between cfgadm commands and Solaris hotplug framework commands. This phase of work will port PCI SHPC and PCI Express hotplug controller drivers to the new hotplug framework interfaces. The current cfgadm ioctl implementation in these drivers will be removed. But all the existing cfgadm PCI functionality will be supported via the new cfgadm shp plugin. In the future (not in this case's scope), when some other hotplug controller drivers are also ported to Solaris hotplug framework, the cfgadm shp plugin can also be used to supported those drivers (may need some updates to shp plugin for hardware specific commands support).

No interfaces are impacted by the new plugin, it is just one more cfgadm plugin. And the existing pci plugin is kept to continue supporting legacy PCI hotplug controllers (the ones except PCI SHPC and PCI Express hotplug controllers).

3. Interfaces

This section focus on interfaces exported by Solaris hotplug framework in the kernel. For interfaces in userland stack, refer to userland documents of this case.

3.1 modctl Commands

modctl is an existing system call in Solaris. This document adds a new command and also a couple of sub-commands to it. Userland programs call modctl() to issue the new command for hotplug operations. The syntax of the modctl is as following (copied from uts/common/os/modctl.c):

int modctl(int cmd, uintptr_t a1, uintptr_t a2, uintptr_t a3, uintptr_t a4, uintptr_t a5);

Argument *a1* is usually taken as a sub-comand, and *a2* to *a5* are arguments.

This document introduces a new modctl command and its sub commands as following. Solaris hotplug framework implements them.

- MODHPOPS

  The new modctl command to be added to uts/common/sys/modctl.h for hotplug operations.

  - MODHPOPS_CHANGE_STATE

    Sub-command for changing hotplug state of a connection. Argument list:

    int modctl(MODHPOPS, MODHPOPS_CHANGE_STATE, char *path, char *cn_name, int state);

    When MODHPOPS_CHANGE_STATE sub-command is received, the framework tries to operate the connection specified by *path* and *cn_name,* and change connection state to the target *state.* The modctl() returns success (0) if the target state is reached. Otherwise, it returns an errors code.

  - MODHPOPS_CREATE_PORT

Sub-command for creating a port. Argument list:

int modctl(MODHPOPS, MODHPOPS_CREATE_PORT, char *path, char *port_name);

When MODHPOPS_CREATE_PORT command is received, the framework creates a new port with name *port_name* under dev_node specified by *path*. It returns success (0) if the port is successfully created. Otherwise, it returns an errors code.

- MODHPOPS_REMOVE_PORT

  Sub-command for removing a port. Argument list:

  int modctl(MODHPOPS, MODHPOPS_REMOVE_PORT, char *path, char *port_name);

  When MODHPOPS_REMOVE_PORT command is received, the framework removes the port specified by *path* and *port_name*. The modctl() returns success (0) if the port is successfully removed. Otherwise, it returns an errors code.

- MODHPOPS_BUS_SET

  Sub-command for setting a bus specific property related with a connection. Argument list:

  int modctl(MODHPOPS, MODHPOPS_BUS_SET, char *path, char *cn_name,

  ddi_hp_property_t *prop, ddi_hp_property_t *result);

  When MODHPOPS_BUS_SET command is received, the framework copyin the property and pass it to the hotplug controller driver. The latter parses unpack the property and set it the connection specified by *path* and *cn_name*. The modctl() returns success (0) if the property is successfully set. Otherwise, it returns an error code.

- MODHPOPS_BUS_GET

  Sub-command for getting a bus specific property related with a connection. Argument list:

  int modctl(MODHPOPS, MODHPOPS_BUS_GET, char *path, char *cn_name, ddi_hp_property_t *prop,

  ddi_hp_property_t *result);

  When MODHPOPS_BUS_GET command is received, the framework get the value of the property (specified by *prop*) from the connection specified by *path* and *cn_name*, and then just pass it up to userland via *result*. The modctl() returns success (0) if the property's value is successfully get. Otherwise, it returns an error code.

The data structure ddi_hp_property_t mentioned in the argument list for sub-commands MODHPOPS_BUS_SET and MODHPOPS_BUS_GET is defined as following:

```
typedef struct ddi_hp_property {
        char *nvlist_buf;
        size_t buf_size;
} ddi_hp_property_t
```

The property's name and value can be packed into a nvlist_t(9f) buffer specified by *nvlist_buf,* and the buffer's size is *buf_size.*

3.2 Libdevinfo interfaces

To get information of the connections in a system (including connection name, current state, etc.), new libdevinfo interfaces are added. More details see userland documents in this case directory.

3.3 Hotplug Controller Driver Interfaces

Solaris hotplug framework defines some interfaces for hotplug controller drivers. Via these interfaces, the framework and drivers are able to interact with each other.

3.3.1 ndi_hp_register() & ndi_hp_unregister()

Hotplug controller drivers call these interfaces which are implemented by Solaris hotplug framework. Also refer to Figure 1.

ndi_hp_register()

Synopsis

int ndi_hp_register(dev_info_t *dip, ddi_hp_cn_info_t *cn_info);

Arguments

dip	parent node of the connection.
cn_info	pointer to the structure of connection information.

Return Values

NDI_SUCCESS	operation succeed.
NDI_FAILURE	operation failed.

And other NDI_* values to specify an error type on operation failure case.

Context

This function can be called from user and kernel context.

Hotplug controller drivers call this interface to register a new connection to the framework, that is, to create the connection data structure on a nexus node.

The following structure is input as *cn_info* argument. It specifies the connection to be registered.

```
typedef struct ddi_hp_cn_info {
        char                *cn_name;           /* Connection name */
        int                 cn_connector_num;   /* Connector number */
        int                 cn_port_num;        /* Port number */
        ddi_hp_cn_type_t    cn_type;        /* Connection type: PCI, ... */
        char                *cn_type_str;   /*  The string for the type of the connection. */
        ddi_hp_cn_state_t   cn_state;       /* Hotplug state of the connection */
        dev_info_t          *cn_child;      /* The device occupying this port; NULL if a connector. */
        time32_t            cn_last_change; /* The time when the last state change happen */
} ddi_hp_cn_info_t;
```

cn_name is a string name of a connection, refer to section 2.4 for details.

cn_connector_num is a digital number of a connector. It is unique among the sibling connectors. It is given by bus software (hotplug controller driver), usually derived from hardware properties, e.g., the hardware slot number set by platform manufacturer. If a ddi_hp_cn_info_t structure is for a connector, its *cn_connector_num* is set to the connector's number. If a ddi_hp_cn_info_t structure is for a port, its *cn_connector_num* is set to the port's dependent connector's number; if the port has no dependent connector, then *cn_connector_num* is set to -1.

cn_port_num is a digital number of a port. It is unique among the sibling ports. It is given by bus software (hotplug controller driver) according to bus industry standards. For example, for a port under a PCI nexus node, the port number is encoded from PCI device number and function number. If a ddi_hp_cn_info_t structure is for a port, its *cn_port_num* is set to the port's number. If a ddi_hp_cn_info_t structure is for a connector, its *cn_port_num* is set to -1.

cn_type is for the connection types, see ddi_hp_cn_type_t below. It is supposed that each hotplug controller driver will register their own types to the framework.

```
typedef enum {
        DDI_HP_CN_TYPE_PORT = 0x1,          /* Software port for virtual hotplug.  */
```

```
        DDI_HP_CN_TYPE_PCI    = 0x2,        /* PCI SHPC slot */
        DDI_HP_CN_TYPE_PCIE   = 0x3         /* PCI Express slot */
    } ddi_hp_cn_type_t;
```

cn_type_str is the description string for *cn_type*, e.g., "PCI SHPC Slot", "PCI Express Slot", “Virtual Hotplug Port”, etc. These type strings can be exposed to userland and end users.

cn_state is the hotplug state of the connection. ddi_hp_cn_state_t is enumeration type for the states, see section 2.5 for details.

cn_child points to the child of a port. It is NULL if the port is at state “Port-Empty” or the ddi_hp_cn_info_t structure is for a connector.

cn_last_change tells the time when the last state change happen.

ndi_hp_unregister()

Synopsis

int ndi_hp_unregister(dev_info_t *dip, char *cn_name);

Arguments

dip	device node where the connection exists.
cn_name	name of the connection.

Return Values

NDI_SUCCESS	operation succeed.
NDI_FAILURE	operation failed.

And other NDI_* values to specify an error type on operation failure case.

Context

This function can be called from user and kernel context.

Hotplug controller drivers call this interface to un-register an existing connection from the framework, that is, to remove the connection data structure from a nexus node.

3.3.2 ndi_hp_state_change_req()

Hotplug controller drivers call this interface to submit a state change request to hotplug framework.

Synopsis

int ndi_hp_state_change_req(dev_info_t *dip, char *cn_name, ddi_hp_cn_state_t state, int flag);

Arguments

dip	device node where the connection exists.
cn_name	name of the connection.
State	target state
flag	DDI_HP_REQ_SYNC, return until the state change is done.
	DDI_HP_REQ_ASYNC, return once the state change request is submitted.

Return Values

NDI_SUCCESS	operation succeed.
NDI_FAILURE	operation failed.

And other NDI_* values to specify an error type on operation failure case.

Context

This function can be called from user and kernel context. If with flag DDI_HP_REQ_ASYNC, it can also

be called from interrupt context.

See section 2.7 for more about the usage of this interface.

3.3.3 ndi_hp_walk_cn()

Hotplug controller drivers call this interface to walk the connections linked to a device node. The connections are either the ports for the device node's children or the connectors at the device node.

Synopsis

void ndi_hp_walk_cn(dev_info_t *dip, int (*f)(ddi_hp_cn_info_t *, void *), void *arg);

Arguments

dip	device node where the connection exists.
(*f)()	walk function.
arg	argument to the walk function.

Context

This function can be called from user and kernel context. It should not be called from interrupt context.

3.3.4 bus_hp_op

bus_hp_op is a new bus_op added to struct bus_ops by this case. Solaris hotplug framework calls this interface which is implemented by hotplug controller drivers. Also refer to Figure 1.

```
struct bus_ops  {
        ...
+       /*
+        * NOTE: the following busop entry point is available with version
+        * 10 or greater.
+        */
+       int     (*bus_hp_op)(dev_info_t *dip, char *cn_name,
+                       ddi_hp_op_t op, void *arg, void *result);
};
```

Arguments

dip	parent node of the connection.
cn_name	name of the connection.
op	operations to be done for the connection.
arg	additional data to specify the operation.
result	operation result.

Return Values

DDI_SUCCESS	operation succeed.
DDI_FAILURE	operation failed.

And other DDI_* values to specify an error type on operation failure case.

The operations specified by *op* are defined as following:

```
typedef enum {
        DDI_HPOP_CN_GET_STATE = 1,      /* Get the current connection state */
        DDI_HPOP_CN_CHANGE_STATE,       /* Change connection state */
        DDI_HPOP_CN_PROBE,              /* Probe a connection */
        DDI_HPOP_CN_UNPROBE,            /* Unprobe a connection */
        DDI_HPOP_CN_GET_PROPERTY,       /* Get properties of connection */
        DDI_HPOP_CN_SET_PROPERTY,       /* Set properties of connection */
        DDI_HPOP_CN_CREATE_PORT,        /* Create a port for virtual hotplug */
        DDI_HPOP_CN_REMOVE_PORT         /* Remove an empty port */
} ddi_hp_op_t;
```

For each *op* in ddi_hp_op_t, the following tell the details of arguments *arg* and *result* in (*bus_hp_op)().

- DDI_HPOP_CN_GET_STATE
  - arg NULL
  - result ddi_hp_cn_state_t * /* Current connection state */
- DDI_HPOP_CN_CHANGE_STATE
  - arg ddi_hp_cn_state_t * /* Target connection state */
  - result ddi_hp_cn_state_t * /* Result connection state */
- DDI_HPOP_CN_PROBE
  - arg NULL
  - result NULL
- DDI_HPOP_CN_UNPROBE
  - arg NULL
  - result NULL
- DDI_HPOP_CN_GET_PROPERTY
  - arg ddi_hp_property_t * /* Specify the property name to get */
  - result ddi_hp_property_t * /* The property name and value to return */
- DDI_HPOP_CN_SET_PROPERTY
  - arg ddi_hp_property_t * /* Specify the property name and value to set */
  - result ddi_hp_property_t * /* Return connection specific information */
- DDI_HPOP_CN_CREATE_PORT
  - arg NULL
  - result NULL
- DDI_HPOP_CN_REMOVE_PORT
  - arg NULL
  - result NULL

3.4 Interface Changes for PCI Configurator

PCI configurator is an existing Solaris kernel module (/kernel/misc/[amd64/]pcicfg or /kernel/misc/sparcv9/pcicfg.e).

This project modifies the interfaces of the existing PCI configurator to support virtual hotplug. The existing interfaces can support physical hotplug only.

To discover a PCI board hotplugged to a PCI slot, PCI configurator is called by hotplug controller drivers to probe the device in the slot and also allocate resources (e.g., I/O or memory addresses, bus numbers, etc.) to the newly plugged device. And then, it reprograms the BARs (Base Address Register) of the PCI device to reflect the resource allocation. After that, the driver goes ahead to setup dev_info properties for the device. The existing pcicfg interfaces are consolidation private (see PSARC/1999/686). They are listed as following:

```
int pcicfg_configure(dev_info_t *devi, uint_t device);

int pcicfg_unconfigure(dev_info_t *devi, uint_t device);
```

There are mainly two issues of the above interfaces if we want to support virtual hotplug for PCI devices.

- Need one more argument to specify a PCI function.

  A PCI device can have more than one PCI functions, and each function is corresponding to a dev_info node in Solaris device tree. Virtual hotplug supports the granularity of each dev_info node. In the example in section 2.5.5, in step 4, to discover a specific NIC function on the PCI NIC board, the software need to specify both PCI device number and function number when call pcicfg_configure(). It is the similar case for pcicfg_unconfigure().

- Need flags to specify read-only probe/unprobe and also some other miscellaneous behaviors.

  For instance, in the example in section 2.5.5, in step 4, in a non-root domain, all devices to be discovered should already have resources allocated and hardware registers programmed. pcicfg_configure() is called to read the registers for getting the information of allocated resources and then setup dev_info node for the device. It is different with regular probe because no hardware registers will be programmed.

The new pcicfg interfaces are listed as following:

```
int pcicfg_configure(dev_info_t *devi, uint_t device, uint_t function, pcicfg_flags_t flags);

int pcicfg_unconfigure(dev_info_t *devi, uint_t device, uint_t function, pcicfg_flags_t flags);

typedef enum pcicfg_flags {
        PCICFG_FLAG_READONLY_PROBE = 0x1,       /* Read-only probe. */
        PCICFG_FLAG_READONLY_UNPROBE = 0x2,     /* Read-only unprobe. */
        PCICFG_FLAG_ARI_PROBE = 0x4             /* Probe ARI device functions */
} pcicfg_flags_t;
```

The details for the flags:

- PCICFG_FLAG_READONLY_PROBE

  It can be set when calling pcicfg_configure(). It does not program hardware registers when probe device. All the current PCI resource assignments will be kept.

- PCICFG_FLAG_READONLY_UNPROBE

  It can be set when calling pcicfg_unconfigure(). It does not program hardware registers when unprobe device.

- PCICFG_FLAG_ARI_PROBE

  It can be set when calling pcicfg_configure(). It enables PCI Alternative Routing-ID Interpretation (ARI) and probes the PCI device functions with function numbers which are valid only in ARI enabled case.

There might be more flags needed in the future (out of scope of this case).

3.5 Interface Tables

Table 2 summarizes all of the exported interfaces by this document.

Table 2. Exported Interfaces

Interface	Stability	Comments
MODHPOPS	Project private	Modctl command for hotplug operations
MODHPOPS_CHANGE_STATE	Project private	Modctl sub-command for changing hotplug state
MODHPOPS_CREATE_PORT	Project private	Modctl sub-command for creating a port
MODHPOPS_REMOVE_PORT	Project private	Modctl sub-command for removing a port
MODHPOPS_BUS_SET	Project private	Modctl sub-command for setting bus specific hotplug properties.
MODHPOPS_BUS_GET	Project private	Modctl sub-command for getting bus specific hotplug properties.
ddi_hp_property_t	Project private	Structure used to get/set bus specific hotplug properties.
ndi_hp_register()	Consolidation private	Register a connection to hotplug framework
ndi_hp_unregister()	Consolidation private	Unregister a connection from hotplug framework
ndi_hp_state_change_req()	Consolidation private	Submit a state change request to hotplug framework
ndi_hp_walk_cn();	Consolidation private	Walk the connections linked to a device node
ddi_hp_cn_info_t	Consolidation private	Structure for the information of a connection
ddi_hp_cn_state_t	Consolidation private	enum for the hotplug states of a connection
BUSO_REV_10	Consolidation private	bus_ops busops_rev
bus_hp_op()	Consolidation private	Add a bus_ops entry for hotplug operations
ddi_hp_op_t	Consolidation private	Hotplug operations
pcicfg_configure()	Consolidation private	Update the existing interface for probing and configuring a connection.
pcicfg_unconfigure()	Consolidation private	Update the existing interface for unprobing and unconfiguring a connection.
pcicfg_flags_t	Consolidation private	New flags for the argument to pcicfg_configure() and pcicfg_unconfigure().

The imported interfaces by this document include:

- pcicfg (PSARC/1999/686). Consolidation Private; It is the interface for configure/unconfigure PCI devices.
- modctl(). modctl is a system call with a long history and the project team does not find a previous ARC case defining it. So the stability level of it is not clear. But the new modctl commands added in recently years are usually Consolidation Private or Project Private, for example, PSARC/2007/290. The new modctl commands introduced by this project are all Project Private, see the above exported interface table.