Proceedings of the Linux Symposium

Volume Two

July 20nd–23th, 2005
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Stephanie Donovan, *Linux Symposium*

Review Committee

Gerrit Huizenga, *IBM*
Matthew Wilcox, *HP*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Matt Domsch, *Dell*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

PCI Express Port Bus Driver Support for Linux

Tom Long Nguyen, Dely L. Sy, & Steven Carbonari
*Intel® Corporation**
{tom.l.nguyen, dely.l.sy, steven.carbonari}@intel.com

Abstract

PCI Express®[1] is a high performance general purpose I/O Interconnect defined for a wide variety of computing and communication platforms. It defines PCI Express Ports and switches to provide a fabric based point-to-point topology. PCI Express categorizes PCI Express Ports into three types: the Root Ports, the Switch Upstream Ports, and the Switch Downstream Ports. Each PCI Express Port can provide up to four distinct services: native hot-plug, power management, advanced error reporting, and virtual channels[1][3]. To fit within the existing Linux®[2] PCI Driver Model but provide a clean and modular solution, in which each service driver can be built and loaded independently, requires the PCI Express Port Bus Driver architecture. The PCI Express Port Bus Driver initializes all services and distributes them to their corresponding service drivers. This paper is targeted toward kernel developers and architects interested in the details of enabling service drivers for PCI Express Ports. The i386 Linux implementation will be used as a reference model to provide insight into the implementation of the PCI Express Port Bus Driver and specific service drivers like the advanced error reporting root service driver and the native hot-plug root service driver.

1 Introduction

The Linux PCI Driver Model restricts a device to a single driver. Drivers in Linux are loaded based off the PCI Device ID and function. Once a driver is loaded, no other drivers for that device can be loaded[2]. Referring to Figure 1, if the Root Port hot-plug driver is loaded first, it claims the Root Port device. The Linux PCI Driver Model therefore prevents the support of multiple services per PCI Express Port using individual service drivers.

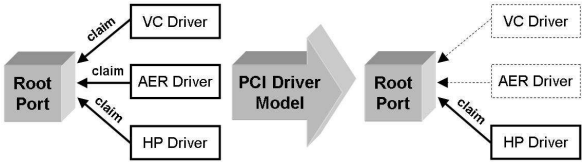

Figure 1: Service Drivers under the Linux PCI Driver Model

A PCI Express Port may have multiple distinct services operating independently. A PCI Express Port is not required to support all services, so some PCI Express Ports within a PCI Express hierarchy may support none, some or all the services. A possible solution is to implement a single driver to handle all services

*Intel is a trademark or registered trademark of Intel Corporation in the United States, other countries, or both. This work represents the view of the authors and does not necessarily represent the view of Intel.

[1]PCI Express is a trademark of the Peripheral Component Interchange Special Interest Group (PCI-SIG)

[2]Linux is a registered trademark of Linus Torvalds

per PCI Express Port. However, this solution would lack the ability to have each service built and loaded independently from each other, preventing extensibility for addition of future services and the ability to have a service driver loaded on more than one PCI Express Port. Separate service drivers are required to support addition of new features and loading of services based on the PCI Express Port capabilities.

To support multiple drivers per device without changing the existing Linux PCI Driver Model requires a new architecture that fits within the existing Linux PCI Driver Model but provides the flexibility required to support multiple service drivers per PCI Express Port. As shown in Figure 2, the PCI Express Port Bus Driver (PBD)[5] fits into the existing Linux PCI Driver Model while providing an interface to allow multiple independent service drivers to be loaded for a single PCI Express Root Port. The PBD acts as a service manager that owns all services implemented by the Ports. Each of these services is then distributed and handled by a unique service driver. The PBD achieves the following key advantages:

- Allows multiple service drivers to run simultaneously and independently from each other and to service more than one PCI Express Port.
- Allows service drivers to be designed and implemented in a modular fashion.
- Centralizes management and distribution of resources of the PCI Express Port devices to requested service drivers.

This paper describes the PCI Express Port Bus Driver architecture. Following the port bus driver architecture are two examples of service drivers. The first example is the advanced error reporting service driver that was designed to work with the port bus architecture. The second example is the hot-plug service driver that was originally designed as an independent driver then converted to a service driver to operate with the Port Bus Driver. Lastly, an overview of the impact to device drivers and future service drivers is outlined.

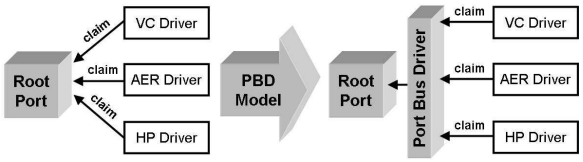

Figure 2: Service Drivers under the PBD

2 PCI Express Port Bus Driver

2.1 PCI Express Port Topology

To understand the Port Bus Driver architecture, it helps to begin with the basics of PCI Express Port topology. Figure 3 illustrates two types of PCI Express Port devices: the Root Port and the Switch Port. The Root Port originates a PCI Express Link from a PCI Express Root Complex. The Switch Port, which has its secondary bus representing switch internal routing logic, is called the Switch Upstream Port. The Switch Port which is bridging from switch internal routing buses to the bus representing the downstream PCI Express Link is called the Switch Downstream Port[1].

Each PCI Express Port device can be implemented to support up to four distinct services: native hot plug (HP), power management event (PME), advanced error reporting (AER), virtual channels (VC). The PCI Express services discussed are optional, so in any given PCI Express hierarchy a port may support none, some, or all of the services.

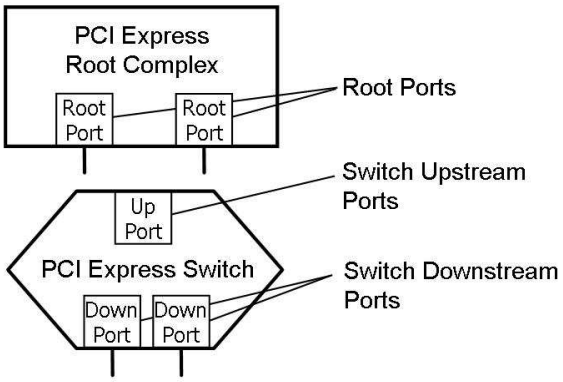

Figure 3: PCI Express Port Topology

2.2 PCI Express Port Bus Driver Architecture

The design of the PCI Express Port Bus Driver achieves a clean and modular solution in which each service driver can be built and loaded independently from each other. The PCI Express Port Bus Driver serves as a service manager that loads and unloads the service drivers accordingly, as illustrated in Figure 4.

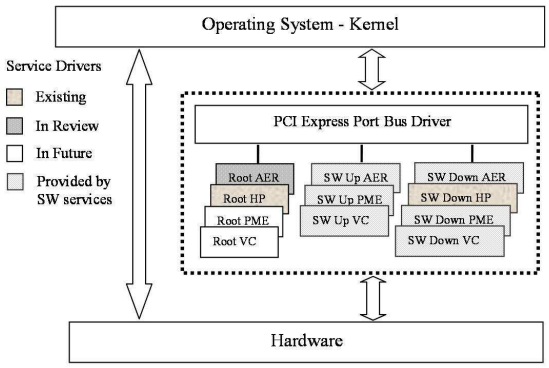

Figure 4: PCI Express Port Bus Driver System View

The PCI Express Port Bus Driver is a PCI-PCI Bridge device driver, which attaches to PCI Express Port devices. For each PCI Express Port device, the PCI Express Port Bus Driver searches for all possible services, such as native HP, PME, AER, and VC, implemented by PCI Express Port device. For each service found, the PCI Express Port Bus Driver creates a corresponding service device, named pcieXY where X indicates the PCI Express Port type and Y indicates the PCI Express service type as described in Table 1, and then registers this service device into a system device hierarchy. Figure 5 shows an example of how the PCI Express Port Bus Driver creates service devices on a system populated with two Root Port devices, one Switch Upstream Port device, and two Switch Downstream Port devices.

Port Type (X)	Service Type (Y)	Service Entity Description (pcieXY)
0	0	PME service on PCI Express Root Port (PMErs)
0	1	AER service on PCI Express Root Port (AERrs)
0	2	HP service on PCI Express Root Port (HPrs)
0	3	VC service on PCI Express Root Port (VCrs)
1	0	PME service on PCI Express Switch Upstream Port (PMEus)
1	1	AER service on PCI Express Switch Upstream Port (AERus)
1	2	Not a supported PCI Express configuration
1	3	VC service on PCI Express Switch Upstream Port (VCus)
2	0	PME service on PCI Express Switch Downstream Port (PMEds)
2	1	AER service on PCI Express Switch Downstream Port (AERds)
2	2	HP service on PCI Express Switch Downstream Port (HPds)
2	3	VC service on PCI Express Switch Downstream Port (VCds)

Table 1: Service Entity Description

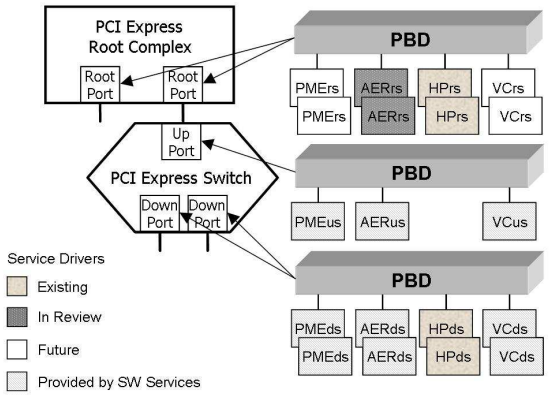

Figure 5: Service Devices in a PCI Express Port Bus Driver Architecture

Once service devices are discovered and added in the system device hierarchy, a service driver is loaded accordingly if it registers its service with the PCI Express Port Bus Driver. The PCI Express Port Bus Driver provides an interface, named `pcie_port_service_register`, to allow a service driver to register its service[4]. The registration enables the user

to configure services during kernel configuration regardless of HW support. It enables debugging and adding of new services in a modular fashion. When a service driver calls `pcie_port_service_register`, the PCI Express Port Bus Driver loads a service driver by invoking the PCI subsystem, which walks through a system device hierarchy for a service match. If the port bus finds a match, it loads a service driver for that service device.

In addition, the PCI Express Port Bus Driver provides `pcie_port_service_unregister`, to undo the effects of calling function `pcie_port_service_register` [4]. Note that a service driver should always call `pcie_port_service_unregister` when a service driver is unloading.

2.3 The Service Driver

To maintain modularity in the PCI Express Port Bus Driver design, individual service drivers are required. In some cases a driver may already exist for a PCI Express Port. In these instances the driver must be ported to the service driver to allow other service drivers to load on the PCI Express Port. To port drivers to service drivers, the following three basic steps are required. Refer to Sections 3.1.1 to 3.1.3 for a specific example.

- Specify service ID. The PCI Express Port Bus Driver defines the data structure of service ID similar to the data structure of `pci_device_id` except with two additional fields: the `port_type` and `service_type` fields as described in Table 1. Note that failure to specify a correct service ID will prevent the port bus from loading a service driver.
- Initialize service driver. The PCI Express Port Bus Driver defines the data structure of service driver similar to the `pci_driver` data structure. The pointer to the `pci_dev` data structure is replaced with a pointer to the `pcie_device` data structure in each callback function.
- Call `pcie_port_service_register` instead `pci_register_driver`.

Once a service driver is loaded, a service driver should always configure and initialize its own capability structure and required IOs to operate normally without any support from the PCI Express Port Bus Driver. However, a service driver is prohibited from doing the following:

- Switch the interrupt mode on a device. The interrupt mode can be INTx legacy, MSI or MSI-X. A service driver should always use the assigned service IRQ to call `request_irq` to have its software interrupt service routine hookup. Note that the assigned service IRQ may be shared among service drivers; therefore, a service driver should always treat this assigned service IRQ as shared interrupt.
- Access resources that are not directly required by the service. For example, the advanced error reporting service driver is prohibited from accessing any configuration registers other than the Advanced Error Reporting Capability structure. A service driver uses the `port` pointer, a member of the `pcie_device` data structure defined by PBD, to access PCI configuration and memory mapped IO space.
- Call `pci_enable_device` and `pci_set_master` functions. This is no longer necessary because these functions now get called by the PCI Express Port Bus Driver.

2.4 Resource Allocation and Distribution

Service drivers must adhere to the guidelines in this document to deal with resource allocation and distribution. Since all service drivers of a PCI Express Port device are allowed to run simultaneously, a decision of which driver (Port Bus Driver vs. service driver) owns which resource is described in Sections 2.4.1 to 2.4.3. These resources include the MSI capability structure, the MSI-X capability structure, and PCI IO resources.

2.4.1 The MSI Capability Structure

The MSI capability structure enables a device software driver to call `pci_enable_msi` to request an MSI based interrupt. Once MSI is enabled on a device, it stays in this mode until a device driver calls `pci_disable_msi` to return to INTx emulation. Since each service driver runs independently from each other, changing the interrupt mode on the PCI Express Port by an individual service driver may result in unpredictable behavior. Each service driver is therefore prohibited from calling these APIs. The PCI Express Port Bus Driver is responsible for determining the interrupt mode and assigning the service IRQ to each service device accordingly. A service driver must use its service vector when calling `request_irq`/`free_irq`.

2.4.2 The MSI-X Capability Structure

Similar to MSI a device driver for an MSI-X capable device can call `pci_enable_msix` to request MSI-X interrupts. The key difference is that the MSI-X capability structure enables a PCI Express Port device to generate multiple messages. Managing multiple MSI-X vectors is handled by the PCI Express Port Bus driver. The PCI Express Port Bus Driver is responsible for determining the interrupt mode transparent to the service drivers. A service driver must use its service vector when calling `request_irq`/`free_irq`.

If a PCI Express Port device supports MSI-X, the PCI Express Port Bus Driver will request the number of MSI-X messages equal to the number of supported services for the device. This allows each service to have it own hardware interrupt resource independently generated from other services.

2.4.3 PCI IO Resources

PCI IO resources include PCI memory/IO ranges and PCI configuration registers are assigned by BIOS during boot. For PCI memory/IO ranges, the service driver is responsible for initializing its PCI memory/IO maps during service startup. There is possibly where the PCI memory/IO ranges are shared. If this occurs, each service driver is responsible for mapping its PCI memory/IO regions without overstepping on resources of others. The PCI Express Port Bus Driver does not arbitrate access to the regions and assumes service drivers to be well behaved.

For PCI configuration registers, each service runs PCI configuration operation on its own capability structure except the PCI Express capability structure, in which the Device Control register and the Root Control register have unique control bits assigned to AER service and PME service. A read-modify-write should always be handled by the AER/PME service driver. Again this paper assumes that all service drivers are responsible for not overstepping on resources of others.

3 PCI Express Advanced Error Reporting Root Service Driver

PCI Express error signaling can occur on the PCI Express link itself or on behalf of transactions initiated on the link. PCI Express defines the Advanced Error Reporting capability, which is implemented with the PCI Express Advanced Error Reporting Extended Capability Structure, to allow a PCI Express component (agent) to send an error reporting message to the Root Port. The Root Port, a host receiver of all error messages associated with its hierarchy, decodes an error message into an error type and an agent ID and then logs these into its PCI Express Advanced Error Reporting Extended Capability Structure. Depending on whether an error reporting message is enabled in the Root Error Command Register, the Root Port device generates an interrupt if an error is detected[1]. The PCI Express advanced error reporting service driver is implemented to service AER interrupts generated by the Root Ports[6].

Once the PCI Express advanced error reporting service driver is loaded, it claims all AER Root service devices in a system device hierarchy, as shown in Figure 6. For each AERrs service device, the advanced error reporting service driver configures its service device to generate an interrupt when an error is detected. For each detected error, the advanced error reporting service driver performs the followings[6]:

- Gather comprehensive error information.
- Guide error recovery associated with the hierarchy in question based on the comprehensive error information gathered.
- Report error to user in a format with more precise what error type and severity.

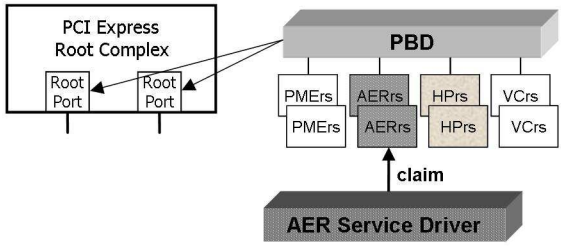

Figure 6: AER Root Service Driver

3.1 Register AER Service

The advanced error reporting service driver is implemented based on the service driver framework as defined in Section 2.3. Sections 3.1.1 to 3.1.3 below illustrate how the advanced error reporting service driver follows three basic steps as required.

3.1.1 Specify AER Service ID

Since the PCI Express advanced error reporting service driver is implemented to serve only the Root Ports, the data structure of AER service ID is defined below[7]:

```
static struct pcie_port_service_id aer_id[]={{
  .vendor = PCI_ANY_ID,
  .device = PCI_ANY_ID,
  .port_type = PCIE_RC_PORT,
  .service_type = PCIE_PORT_SERVICE_AER,
  }, {}
};
```

3.1.2 Initialize AER Service Driver

Once the AER service ID is defined, the advanced error reporting service driver initializes the service callbacks as defined in the `pcie_port_service_driver` data structure. The data structure of service callbacks is defined below[7]:

```
static struct pcie_port_service_driver aerdrv={
  .name = "aer",
  .id_table = &aer_id[0],

  .probe = aer_probe,
  .remove = aer_remove,

  .suspend = aer_suspend,
  .resume = aer_resume,
};
```

3.1.3 Calling `pcie_port_service_register`

The final step in initialization of the advanced error reporting service driver is calling function `pcie_port_service_register` to register AER service with the PBD. During driver initialization, the module routine is called for initialization when the kernel calls the advanced error reporting service driver. Calling `pcie_port_service_register`/`pcie_port_service_unregister` should always be done in `module_init`/`module_exit` as depicted below[7]:

```
static int __init aer_service_init(void)
{
  return pcie_port_service_register(&aerdrv);
}

static void __exit aer_service_exit(void)
{
  pcie_port_service_unregister(&aerdrv);
}

module_init(aer_service_init);
module_exit(aer_service_exit);
```

Figure 7 depicts the state diagram once the advanced error reporting service driver's module routine is called.

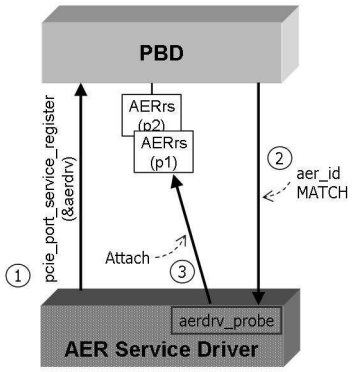

Figure 7: State Diagram of Registering AER Service with PBD

4 PCI Express Native Hot-Plug Service Driver

The PCI Express Hot-Plug standard usage model is derived from the standard usage model defined in the PCI Standard Hot-Plug Controller and Subsystem Specification, Rev. 1.0[8].

4.1 PCI Express Native Hot Plug Features

PCI Express Native Hot-Plug features are:

- Root ports and downstream ports of switches are hot-pluggable ports in PCI Express hierarchy.
- Interrupt driven hot plug events will result in hot-plug interrupts.
- Hot plug registers are part of the PCI Express Capability register set; hot-plug operations are invoked by writing to these registers.
- Based on SHPC usage model, but not the bus centric SHPC register set.

4.2 Porting the PCI Express Hot-Plug Driver to a Service Driver

As mentioned in Section 2.2, the PCI Express Port Bus Driver provides a mechanism for a

service driver to register its service. If the requested service is found in a service device hierarchy, the service driver can successfully load. This section focuses on showing what the changes are required to port the PCI Express native hot-plug driver to a service driver.

4.2.1 Registering the Hot Plug Service Driver

The pciehp driver calls `pcie_port_service_register (struct pcie_port_service_driver *driver)` to register its hot-plug service with the PBD. The pciehp driver is responsible for setting up the data structures before calling `pcie_port_service_register`. Below shows the difference in the data structures used when the driver is used as a standard driver or as a service driver[9].

```
+ static struct pcie_port_service_id
+   port_pci_ids[] = {{
+   .vendor = PCI_ANY_ID,
+   .device = PCI_ANY_ID,
+   .port_type = PCIE_ANY_PORT,
+   .service_type = PCIE_PORT_SERVICE_HP,
+   .driver_data = 0,
+ }, { /* end: all zeroes */ }
+ };

- static struct pci_device_id pcied_pci_tbl[]={
- {
-   .class = ((PCI_CLASS_BRIDGE_PCI << 8) |
-   0x00),
-   .class_mask = ~0,
-   .vendor = PCI_ANY_ID,
-   .device = PCI_ANY_ID,
-   .subvendor = PCI_ANY_ID,
-   .subdevice = PCI_ANY_ID,
- }, { /* end: all zeroes */ }
- };
```

4.2.2 Initialize the Hot-Plug Service Driver

Once the HP service ID is defined, the service driver initializes the service callbacks as defined in the `pcie_port_service_driver` data structure. The following shows the changes that need to be made in porting the PCI Express hot-plug driver to a service driver[9].

```
+ static struct pcie_port_service_driver
+   hpdriver_portdrv = {
+   .name = "hpdriver",
+   .id_table = &port_pci_ids[0],
+   .probe = pciehp_probe,
+   .remove = pciehp_remove,
+   .suspend = pciehp_suspend,
+   .resume = pciehp_resume,
+ };

- static struct pci_driver pcie_driver = {
-   .name = "pciehp",
-   .id_table = pcied_pci_tbl,
-   .probe = pcie_probe,
-   .remove = pcie_remove,
- };
```

4.2.3 Calling `pcie_port_service_register` API

The final step in initialization of the HP service driver is calling `pcie_port_service_register` to register HP service with the PBD. The following shows the changes that need to be made in the standalone driver to port it to a service driver[9].

```
static int __init pcied_init(void)
{
     :
+   retval = pcie_port_service_register(
+   &hpdriver_portdrv);
-   retval = pci_register_driver(
-   &pcie_driver);
    :
}

static void __exit pcied_cleanup(void)
{
    :
+   pcie_port_service_unregister(
+   &hpdriver_portdrv);
-   pci_unregister_driver(&pcie_driver);
    :
}
```

Figure 8 depicts the state diagram once HP service driver's module routine is called.

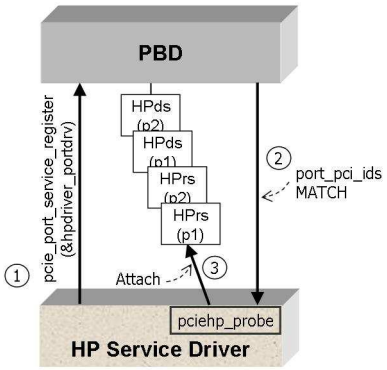

Figure 8: State Diagram of Registering HP Service with PBD

4.2.4 Available Resources

As a service driver, dev->irq is provided by the PCI Express Port Bus Driver and is passed to the pciehp driver. Whether dev->irq is a regular system interrupt, MSI or MSI-X, the PCI Express Port Bus Driver assigns the value to it. The pciehp service driver does not need to call `pci_enable_msi` to request use of MSI/MSI-X if the OS supports that.

5 Impacts to PCI Express Drivers

The Port Bus Driver design does not directly impact existing PCI Express endpoint device drivers. However, a service driver may impact a PCI Express endpoint driver. Additional PCI Express services may require endpoint driver changes to take full advantage of the new functionality. For example, to take full advantage of AER error recovery will require drivers to support the AER callback API. Driver writers for PCI Express components should be well versed with this architecture and evaluate driver impacts as new services (VC or PME) become available.

The Port Bus perspective impacts device drivers for PCI Express Switch components. The PCI Express Port Bus Driver claims all PCI Express Ports in a system device hierarchy, including ports in a PCI Express switch. Switch service drivers must follow the port bus driver framework. Switch vendors can use existing root service drivers as a reference while writing their own service drivers.

When developing a switch service driver the usage model at each level in the PCI Express hierarchy needs to be considered. A service driver for a downstream switch port may be required to provide different functionality than a similar root port service driver. For example, the AER Root service driver cannot be reused `as-is`. The usage model is different. AER Switch service driver should provide error-handling callbacks and AER initialization of the switch, while the AER Root service driver provides the primary mechanism to handle the error recovery process. However, in the case of the hot-plug driver, the same service driver may be used for both the Root Ports and the Switch Downstream Ports because the hot-plug usage model is identical.

6 Conclusion

The design of the PCI Express Port Bus Driver delivers a clean and modular solution to support the multiple features of PCI Express while remaining compatible with the Linux Driver Model. Each feature can have its own software service driver that can be built and loaded as a separate module. In addition when/if future PCI Express features come available or are added to future specification revisions, the PCI Express Port Bus architecture is extensible to support those additions. The PCI Express Port Bus Driver and changes to port the native PCI Express hot-plug driver has been incorporated Linux kernel version 2.6.11. The advanced error reporting service driver is currently under review on the LKML.

7 Acknowledgements

Special thanks to Greg Kroah-Hartman for his contributions to the architecture design of PCI Express Port Bus driver.

References

[1] PCI Express Base Specification Revision 1.1. March 28, 2005. `http://www.pcisig.com/specifications/pciexpress/`.

[2] Linux Device Drivers, 3rd Edition. Publisher: O'Reilly & Associates; 3 edition (February 10, 2005) by Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman.

[3] Renato John Recio. Promises and Reality: Server I/O networks, past, present, and future. In Proceedings of the ACM SIGCOMM Workshop on Network-I/O Convergence: Experience, Lessons, Implications. Pages 163-178, Karlsruhe, Germany, August 2003.

[4] PCIEBUS-HOWTO.txt. Available from: `2.6.11/Documentation`.

[5] PCI Express Port Bus Driver code. Available from: `2.6.11/drivers/pci/pcie`.

[6] PCIEAER-HOWTO.txt, under review. If being accepted: `2.6.x/Documentation`.

[7] PCI Express advanced error reporting driver code, under review. If accepted: 2.6.x/drivers/pci/pcie/aer.

[8] PCI Standard Hot-Plug Controller and Subsystem Specification Revision 1.0. June 20, 2001. `http://www.pcisig.com/specifications/conventional/pci_hot_plug/SHPC_10/`.

[9] PCI Express hot-plug driver code. Available from: `2.6.11/drivers/pci/hotplug`.