The PCI Express Port Bus Driver Guide HOWTO

Author: Tom L Nguyen tom.l.nguyen@intel.com 11/03/2004

About this guide

This guide describes the basics of the PCI Express Port Bus driver and provides information on how to enable the service drivers to register/unregister with the PCI Express Port Bus Driver.

What is the PCI Express Port Bus Driver

A PCI Express Port is a logical PCI-PCI Bridge structure. There are two types of PCI Express Port: the Root Port and the Switch Port. The Root Port originates a PCI Express link from a PCI Express Root Complex and the Switch Port connects PCI Express links to internal logical PCI buses. The Switch Port, which has its secondary bus representing the switch's internal routing logic, is called the switch's Upstream Port. The switch's Downstream Port is bridging from switch's internal routing bus to a bus representing the downstream PCI Express link from the PCI Express Switch.

A PCI Express Port can provide up to four distinct functions, referred to in this document as services, depending on its port type. PCI Express Port's services include native hotplug support (HP), power management event support (PME), advanced error reporting support (AER), and virtual channel support (VC). These services may be handled by a single complex driver or be individually distributed and handled by corresponding service drivers.

Why use the PCI Express Port Bus Driver?

In existing Linux kernels, the Linux Device Driver Model allows a physical device to be handled by only a single driver. The PCI Express Port is a PCI-PCI Bridge device with multiple distinct services. To maintain a clean and simple solution each service may have its own software service driver. In this case several service drivers will compete for a single PCI-PCI Bridge device. For example, if the PCI Express Root Port native hotplug service driver is loaded first, it claims a PCI-PCI Bridge Root Port. The kernel therefore does not load other service drivers for that Root Port. In other words, it is impossible to have multiple service drivers load and run on a PCI-PCI Bridge device simultaneously using the current driver model.

To enable multiple service drivers running simultaneously requires having a PCI Express Port Bus driver, which manages all populated PCI Express Ports and distributes all provided service requests to the corresponding service drivers as required. Some key advantages of using the PCI Express Port Bus driver are listed below:

- Allow multiple service drivers to run simultaneously on a PCI-PCI Bridge Port device.
- Allow service drivers implemented in an independent staged approach.
- Allow one service driver to run on multiple PCI-PCI Bridge Port devices.
- Manage and distribute resources of a PCI-PCI Bridge Port device to requested service drivers.

Configuring the PCI Express Port Bus Driver vs. Service Drivers

Including the PCI Express Port Bus Driver Support into the Kernel

Including the PCI Express Port Bus driver depends on whether the PCI Express support is included in the kernel config. The kernel will automatically include the PCI Express Port Bus driver as a kernel driver when the PCI Express support is enabled in the kernel.

Enabling Service Driver Support

PCI device drivers are implemented based on Linux Device Driver Model. All service drivers are PCI device drivers. As discussed above, it is impossible to load any service driver once the kernel has loaded the PCI Express Port Bus Driver. To meet the PCI Express Port Bus Driver Model requires some minimal changes on existing service drivers that imposes no impact on the functionality of existing service drivers.

A service driver is required to use the two APIs shown below to register its service with the PCI Express Port Bus driver (see section 5.2.1 & 5.2.2). It is important that a service driver initializes the pcie_port_service_driver data structure, included in header file /include/linux/pcieport_if.h, before calling these APIs. Failure to do so will result an identity mismatch, which prevents the PCI Express Port Bus driver from loading a service driver.

pcie_port_service_register

```
int pcie_port_service_register(struct pcie_port_service_driver *new)
```

This API replaces the Linux Driver Model's pci_register_driver API. A service driver should always calls pcie_port_service_register at module init. Note that after service driver being loaded, calls such as pci_enable_device(dev) and pci_set_master(dev) are no

longer necessary since these calls are executed by the PCI Port Bus driver.

pcie_port_service_unregister

```
void pcie_port_service_unregister(struct pcie_port_service_driver *new)
```

pcie_port_service_unregister replaces the Linux Driver Model's pci_unregister_driver. It's always called by service driver when a module exits.

Sample Code

Below is sample service driver code to initialize the port service driver data structure.

```
static struct pcie_port_service_id service_id[] = { {
  .vendor = PCI_ANY_ID,
  .device = PCI_ANY_ID,
  .port_type = PCIE_RC_PORT,
  .service_type = PCIE_PORT_SERVICE_AER,
  }, { /* end: all zeroes */ }
};

static struct pcie_port_service_driver root_aerdrv = {
  .name               = (char *)device_name,
  .id_table   = &service_id[0],

  .probe              = aerdrv_load,
  .remove             = aerdrv_unload,

  .suspend    = aerdrv_suspend,
  .resume             = aerdrv_resume,
};
```

Below is a sample code for registering/unregistering a service driver.

```
static int __init aerdrv_service_init(void)
{
  int retval = 0;

  retval = pcie_port_service_register(&root_aerdrv);
  if (!retval) {
    /*
    * FIX ME
    */
  }
  return retval;
}

static void __exit aerdrv_service_exit(void)
{
  pcie_port_service_unregister(&root_aerdrv);
}

module_init(aerdrv_service_init);
module_exit(aerdrv_service_exit);
```

Possible Resource Conflicts

Since all service drivers of a PCI-PCI Bridge Port device are allowed to run simultaneously, below lists a few of possible resource conflicts with proposed solutions.

MSI and MSI-X Vector Resource

Once MSI or MSI-X interrupts are enabled on a device, it stays in this mode until they are disabled again. Since service drivers of the same PCI-PCI Bridge port share the same physical device, if an individual service driver enables or disables MSI/MSI-X mode it may result unpredictable behavior.

To avoid this situation all service drivers are not permitted to switch interrupt mode on its device. The PCI Express Port Bus driver is responsible for determining the interrupt mode and this should be transparent to service drivers. Service drivers need to know only the vector IRQ assigned to the field irq of struct pcie_device, which is passed in when the PCI Express Port Bus driver probes each service driver. Service drivers should use (struct pcie_device*)dev->irq to call request_irq/free_irq. In addition, the interrupt mode is stored in the field interrupt_mode of struct pcie_device.

PCI Memory/IO Mapped Regions

Service drivers for PCI Express Power Management (PME), Advanced Error Reporting (AER), Hot-Plug (HP) and Virtual Channel (VC) access PCI configuration space on the PCI Express port. In all cases the registers accessed are independent of each other.

This patch assumes that all service drivers will be well behaved and not overwrite other service driver's configuration settings.

PCI Config Registers

Each service driver runs its PCI config operations on its own capability structure except the PCI Express capability structure, in which Root Control register and Device Control register are shared between PME and AER. This patch assumes that all service drivers will be well behaved and not overwrite other service driver's configuration settings.