The anatomy of a PCI/PCI Express kernel driver

Eli Billauer

May 16th, 2011 / June 13th, 2011

This work is released under Creative Common's CC0 license version 1.0 or later. To the extent possible under law, the author has waived all copyright and related or neighboring rights to this work.
Introduction
Overview

- Kernel programming is copying from people knowing better than you.
- Reading PCI/PCIe drivers is being told the solution without understanding the problem.
- We shall try to fill that gap
- Part I: Understanding the hardware: Buses, PCI, PCIe, interrupts
- Part II: Highlights of a PCI/PCIe driver
- Not covered: General kernel hacking practices (character devices, mutexes, spinlocks etc.)
Part I: The hardware
It’s all about wires and voltages

- The Memory bus
- The I/O bus
- The bus clock
- Bus locking / wait states
- The Bridge and Delayed Transaction
- Bursts
- Bus mastering, bus arbitration, DMA
Introduction
Part I: The hardware
The Strategy
Part II: Code highlights

PCI

- Very slow bus in today’s terms (66 MHz)
- Memory space (32/64 bits), I/O space, configuration space
- Bus Mastering (DMA) and bursts
- Bridges
- Vendor ID, Device ID, class, subclass
- pci.ids, lspci and kernel support of a specific device
PCI: Enumeration

- Bus number, Device number, Function
- The Configuration Header
- Automatic allocation of dedicated addresses segments ("BAR addresses") by OS (BIOS on PCs)
- ... and interrupts
- Tons of extensions
- PCIe mimics PCI, so we’ll get to it later...
- Try `lspci -tv` and `setpci`
$ lspci
00:00.0 Host bridge: Intel Corporation Clarksfield/Lynnfield DMI (rev 11)
00:03.0 PCI bridge: Intel Corporation Clarksfield/Lynnfield PCI Express Root Port 1 (rev 11)
00:08.0 System peripheral: Intel Corporation Clarksfield/Lynnfield System Management Registers (rev 11)
00:08.1 System peripheral: Intel Corporation Clarksfield/Lynnfield Semaphore and Scratchpad Registers (rev 11)
00:08.2 System peripheral: Intel Corporation Clarksfield/Lynnfield System Control and Status Registers (rev 11)
00:08.3 System peripheral: Intel Corporation Clarksfield/Lynnfield Miscellaneous Registers (rev 11)
00:10.0 System peripheral: Intel Corporation QPI Link (rev 11)
00:10.1 System peripheral: Intel Corporation QPI Routing and Protocol Registers (rev 11)
00:1a.0 USB Controller: Intel Corporation Ibex Peak USB Universal Host Controller (rev 05)
00:1a.1 USB Controller: Intel Corporation Ibex Peak USB Universal Host Controller (rev 05)
00:1a.2 USB Controller: Intel Corporation Ibex Peak USB Universal Host Controller (rev 05)
00:1a.7 USB Controller: Intel Corporation Ibex Peak USB2 Enhanced Host Controller (rev 05)
00:1b.0 Audio device: Intel Corporation Ibex Peak High Definition Audio (rev 05)
00:1c.0 PCI bridge: Intel Corporation Ibex Peak PCI Express Root Port 1 (rev 05)
00:1c.1 PCI bridge: Intel Corporation Ibex Peak PCI Express Root Port 2 (rev 05)
00:1c.3 PCI bridge: Intel Corporation Ibex Peak PCI Express Root Port 4 (rev 05)
lspci output (more)

00:1d.0 USB Controller: Intel Corporation Ibex Peak USB Universal Host Controller (rev 05)
00:1d.1 USB Controller: Intel Corporation Ibex Peak USB Universal Host Controller (rev 05)
00:1d.2 USB Controller: Intel Corporation Ibex Peak USB Universal Host Controller (rev 05)
00:1d.3 USB Controller: Intel Corporation Ibex Peak USB Universal Host Controller (rev 05)
00:1d.7 USB Controller: Intel Corporation Ibex Peak USB2 Enhanced Host Controller (rev 05)
00:1e.0 PCI bridge: Intel Corporation 82801 PCI Bridge (rev a5)
00:1f.0 ISA bridge: Intel Corporation Ibex Peak LPC Interface Controller (rev 05)
00:1f.2 SATA controller: Intel Corporation Ibex Peak 6 port SATA AHCI Controller (rev 05)
00:1f.3 SMBus: Intel Corporation Ibex Peak SMBus Controller (rev 05)
01:00.0 VGA compatible controller: ATI Technologies Inc RV710 [Radeon HD 4350]
01:00.1 Audio device: ATI Technologies Inc R700 Audio Device [Radeon HD 4000 Series]
02:00.0 SATA controller: JMicron Technologies, Inc. 20360/20363 Serial ATA Controller (rev 02)
02:00.1 IDE interface: JMicron Technologies, Inc. 20360/20363 Serial ATA Controller (rev 02)
03:00.0 Ethernet controller: Realtek Semiconductor Co., Ltd. RTL8111/8168B PCI Express Gigabit Ethernet controller (rev 03)
04:00.0 SATA controller: JMicron Technologies, Inc. 20360/20363 Serial ATA Controller (rev 03)
04:00.1 IDE interface: JMicron Technologies, Inc. 20360/20363 Serial ATA Controller (rev 03)
05:04.0 Modem: ALi Corporation SmartLink SmartPCI563 56K Modem
05:05.0 Ethernet controller: 3Com Corporation 3CSDH0100B-TX 910-A01 [tulip] (rev 31)
$ lspci -tv
-[0000:00]-+-00.0 Intel Corporation Clarksfield/Lynnfield DMI
  +-03.0-[01]----00.0 ATI Technologies Inc RV710 [Radeon HD 4350]
  |    \-00.1 ATI Technologies Inc R700 Audio Device [Radeon HD 4000 Series]
  +-08.0 Intel Corporation Clarksfield/Lynnfield System Management Registers
  +-08.1 Intel Corporation Clarksfield/Lynnfield Semaphore and Scratchpad Registers
  +-08.2 Intel Corporation Clarksfield/Lynnfield System Control and Status Registers
  +-08.3 Intel Corporation Clarksfield/Lynnfield Miscellaneous Registers
  +-10.0 Intel Corporation QPI Link
  +-10.1 Intel Corporation QPI Routing and Protocol Registers
  +-1a.0 Intel Corporation Ibex Peak USB Universal Host Controller
  +-1a.1 Intel Corporation Ibex Peak USB Universal Host Controller
  +-1a.2 Intel Corporation Ibex Peak USB Universal Host Controller
  +-1a.7 Intel Corporation Ibex Peak USB2 Enhanced Host Controller
  +-1b.0 Intel Corporation Ibex Peak High Definition Audio
  +-1c.0-[02]----00.0 JMicron Technologies, Inc. 20360/20363 Serial ATA Controller
  |    \-00.1 JMicron Technologies, Inc. 20360/20363 Serial ATA Controller
  +-1c.1-[03]----00.0 Realtek Semiconductor Co., Ltd. RTL8111/8168B PCI Express Gigabit Ethernet
  +-1c.3-[04]----00.0 JMicron Technologies, Inc. 20360/20363 Serial ATA Controller
  |    \-00.1 JMicron Technologies, Inc. 20360/20363 Serial ATA Controller
  +-1d.0 Intel Corporation Ibex Peak USB Universal Host Controller
  +-1d.1 Intel Corporation Ibex Peak USB Universal Host Controller
  +-1d.2 Intel Corporation Ibex Peak USB Universal Host Controller
  +-1d.3 Intel Corporation Ibex Peak USB Universal Host Controller
  +-1d.7 Intel Corporation Ibex Peak USB2 Enhanced Host Controller
  +-1e.0-[05]----04.0 ALi Corporation SmartLink SmartPCI563 56K Modem
  |    \-05.0 3Com Corporation 3CSOHO100B-TX 910-A01 [tulip]
  +-1f.0 Intel Corporation Ibex Peak LPC Interface Controller
  +-1f.2 Intel Corporation Ibex Peak 6 port SATA AHCI Controller
  \-1f.3 Intel Corporation Ibex Peak SMBus Controller
The ATI card is Vendor/Product ID 1002:954f

```
# lspci -v
(...)
01:00.0 VGA compatible controller: ATI Technologies Inc RV710 [Radeon HD 4350] (prog-if 00 [VGA controller]
  Subsystem: Giga-byte Technology Device 21ac
  Flags: bus master, fast devsel, latency 0, IRQ 50
  Memory at e0000000 (64-bit, prefetchable) [size=256M]
  Memory at fbce0000 (64-bit, non-prefetchable) [size=64K]
  I/O ports at ae00 [size=256]
  [virtual] Expansion ROM at fbc0000 [disabled] [size=128K]
  Capabilities: [50] Power Management version 3
  Capabilities: [58] Express Legacy Endpoint, MSI 00
  Capabilities: [a0] MSI: Enable+ Count=1/1 Maskable- 64bit+
  Capabilities: [100] Vendor Specific Information: ID=0001 Rev=1 Len=010 <?
  Kernel driver in use: radeon
  Kernel modules: radeon
```

```
# lspci -x
(...)
01:00.0 VGA compatible controller: ATI Technologies Inc RV710 [Radeon HD 4350]
  00: 02 10 4f 95 07 04 10 00 00 00 00 03 10 00 80 00
  10: 0c 00 00 e0 00 00 00 00 00 04 00 ce fb 00 00 00 00
  20: 01 ae 00 00 00 00 00 00 00 00 00 00 58 14 ac 21
  30: 00 00 00 00 50 00 00 00 00 00 00 00 00 00 0b 01 00 00
```
I/O Mapped I/O

- It makes sense, doesn’t it?
- But it’s deprecated on new designs (parallel port at 0x0378, anyone?)
- Widely used (on chipsets?), but to be removed from future PCIe spec.
- Delayed Transaction not allowed on I/O writes. (but is on Memory writes) PCI and PCIe alike.
- 16 bits addressing on x86. Depends on architecture.
- PCI allows 32 bits.
- Check out /proc/ioports for your own computer’s mapping.
Memory mapped I/O

- The processor thinks it’s memory, so does the compiler. So they optimize.
- Read prefetching (side effects?)
- Write reordering (Reset and kick-off writes?)
- Cache coherency and flushing
- Read `memory-barriers.txt` in the kernel source’s Documentation.
- `volatile-considered-harmful.txt` and `circular-buffers.txt` are recommended as well.
- IOMMU
- Check out `/proc/iomem` for your own computer’s mapping.
The Interrupt

- A processor will make a jump to a known address more or less immediately
- Physical interrupt lines are limited
- PCI has INTA, INTB, INTC, INTD.
- Interrupt sharing: Everyone answers the phone
- Edge triggered can’t be shared, and double interrupts are missed
- Level triggered can
- MSI: Message Signalled Interrupt. The savior.
- Check out /proc/interrupts to get the nudges.
PCI Express is not PCI-X!
It’s not a bus, it’s a packet network of 32 bit elements
Hotpluggable, energy saving (for laptops...)
Each device has its own dedicated wires
Bus switches as routers
Transaction Layer Packets (TLPs) instead of bus transactions
Posted and non-posted bus requests
Data flow control, error detection and retransmit
Design to look, feel and smell like PCI
Full backward compatibility and interoperability with legacy PCI (including INTx)
PCI express (cont.)

- Lanes: 1x, 2x, 4x, 8x, 16x
- Speeds: PCIe 1.0: 2.5 Gb/s raw bits, 2.0 Gb/s data bits, parallel to 62.5 MHz 32-bit bus...
  - ... in both directions!
- Doubled in PCIe 2.0, and sky is the limit
- Multiple lanes work in sync to transmit 32-bit words faster.
- The specification is wisely designed
- ... and horribly written
The Strategy
How a (sane) PCI/PCIe device works

- The BAR address space (mapped in memory or I/O space) is used for control registers
- The driver allocates buffers in RAM
- The addresses of these buffers are written in control registers
- The device reads and writes from the buffer via DMA
- All this is timed and orchestrated via control registers
- ... and interrupts
- Video cards may take a different strategy.
Part II: Code highlights
static const struct pci_device_id foo_ids[] = {
    {
      .class = 0x000000,
      .class_mask = 0x000000,
      .vendor = 0xdead,
      .device = 0xbeef,
      .subvendor = PCI_ANY_ID,
      .subdevice = PCI_ANY_ID,
      .driver_data = 0 /* Arbitrary data, hence zero */
    },
    { /* End: all zeroes */ }
};
Declarations

Somewhere at the end of the module...

```c
MODULE_DEVICE_TABLE(pci, foo_ids);

static struct pci_driver foo_driver = {
    .name = "foo",
    .id_table = foo_ids,
    .probe = foo_probe,
    .remove = __devexit_p(foo_remove),
};
```
Finishing up the bureaucrats

Still at the end of the module...

```c
static int __init foo_init(void)
{
    return pci_register_driver(&foo_driver);
}

static void __exit foo_exit(void)
{
    pci_unregister_driver(&foo_driver);
}

module_init(foo_init);
module_exit(foo_exit);
```
What we have, what we’ll do

- We’ve told the kernel which device(s) we want to handle
- Kernel will invoke our probe function when meeting one such
- ... in process context
- So for the sake of example, all code from here on belongs to the probe function
- In reality, the real action happens in the character / block / network device’s code
- The operations are shown in a possible real-life order
The probe method

static int __devinit foo_probe(struct pci_dev *pdev,
    const struct pci_device_id *ent)
{
    struct foo_privdata *privdata;
    int rc = 0;
    int i;

    privdata = kzalloc(sizeof(*privdata), GFP_KERNEL);
    if (!privdata) {
        printk(KERN_ERR "foo: Failed to allocate memory.\n");
        return -ENOMEM;
    }

    pci_set_drvdata(pdev, privdata);
Enabling the device

```c
rc = pci_enable_device(pdev);

if (rc) {
    printk(KERN_ERR "foo: pci_enable_device() failed.\n");
    goto no_enable;
}
```
A small sanity check

Checking that BAR 0 is defined and memory mapped:

    if (!(pci_resource_flags(pdev, 0) & IORESOURCE_MEM)) {
        printk(KERN_ERR "foo: Incorrect BAR configuration.\n");
        rc = -ENODEV;
        goto bad_bar;
    }
Taking ownership of a memory region

```
rc = pci_request_regions(pdev, "foo");
if (rc) {
    printk(KERN_ERR "foo: pci_request_regions() failed.\n");
    goto failed_request_regions;
}
```

After this, “foo” will show up in /proc/iomem
Obtaining our impression of the BAR address

Assuming that our device's configuration header requests 128 bytes on BAR 0:

```c
privdata->registers = pci_iomap(pdev, 0, 128);

if (!privdata->registers) {
    printk(KERN_ERR "foo: Failed to map BAR 0.\n");
    goto failed_iomap0;
}

iowrite32(1, &privdata->registers[0x10]);
mmiowb(); /* Some holy water never hurts */
```

registers is pointer to u32.
Reading a bit...

```c
for (i=0; i<16; i++)
    printk(KERN_INFO "foo: Register 0x%x = 0x%x \n", i, ioread32(&privdata->registers[i]));

Also common in the kernel: inp, outp, writel, readl, writew, readw, writeb, readb, ioread8, ioread16, iowrite8, iowrite16
```
Enabling mastership and MSI

pci_set_master(pdev);

/* Set up a single MSI interrupt */
if (pci_enable_msi(pdev)) {
    printk(KERN_ERR "foo: Failed to enable MSI.\n");
    rc = -ENODEV;
    goto failed_enable_msi;
}
if (!pci_set_dma_mask(pdev, DMA_BIT_MASK(64)))
    privdata->dma_using_dac = 1;
else if (!pci_set_dma_mask(pdev, DMA_BIT_MASK(32)))
    privdata->dma_using_dac = 0;
else {
    printk(KERN_ERR "foo: Failed to set DMA mask.\n");
    rc = -ENODEV;
    goto failed_dmamask;
}
int memorder = 2;

privdata->mem = (u32 *) __get_free_pages(GFP_KERNEL | __GFP_DMA | __GFP_ZERO, memorder);

if (!privdata->mem) {
    printk(KERN_ERR "foo: Failed to allocate memory.\n");
    rc = -ENOMEM;
    goto failed_dmamem;
}

printk(KERN_INFO "foo: Got %ld bytes at 0x%x\n",
       PAGE_SIZE * (1 << memorder),
       (u32) privdata->mem);
DMA mapping

dma_mem = pci_map_single(pdev, privdata->mem, PAGE_SIZE * (1 << memorder), PCI_DMA_FROMDEVICE);

if (pci_dma_mapping_error(pdev, dma_mem)) {
    printk(KERN_ERR "foo: Failed to map memory).\n");
    rc = -ENOMEM;
    goto failed_dmamap;
}

dma_mem is a dma_addr_t, which may be a u64 or a u32, depending on the CONFIG_X86_64 or CONFIG_HIGHMEM64G kernel compilation flags.
Preparing for DMA

Tell the device what the DMA address is, and if 64 bits should be used...

```c
iowrite32( (u32) (dma_mem & 0xffffffff),
         &privdata->registers[0x0c]);
iowrite32( ( (u32) ( (dma_mem >> 32) & 0xffffffff) ),
         &privdata->registers[0x0d]);
iowrite32( privdata->dma_using_dac,
         &privdata->registers[0x08]);
mmiowb(); /* Hope the holy water works */
```

**Warning:** Is the PCI device aware of the host's endianess?
Setting up the interrupt handler

```c
init_waitqueue_head(&privdata->waitq);

rc = request_irq(pdev->irq, foo_msi, 0, "foo", privdata);

if (rc) {
    printk(KERN_ERR "foo: Failed to register MSI.\n");
    rc = -ENODEV;
    goto failed_register_msi;
}

waitq is of type wait_queue_head_t
```
A short diversion: The MSI handler

```c
static irqreturn_t foo_msi(int irq, void *data)
{
    struct foo_privdata *privdata = data;

    privdata->flag = 1;
    wake_up_interruptible(&privdata->waitq);

    return IRQ_HANDLED;
}
```

**Important:** On a non-MSI handler, checking that our device really asserted the interrupt is **mandatory**.
privdata->flag = 0;
iowrite32((u32) 1, &privdata->registers[0x09]); /* GO! */

wait_event_interruptible(privdata->waitq,
                        (privdata->flag != 0));

if (privdata->flag == 0) {
    printk(KERN_ERR "foo: Interrupted!\n");
    rc = -ENODEV;
    goto failed_wait_irq;
}

This is correct if we assume no other process waits on the queue, and may clear the flag. It works for the probe() method and if we have a mutex held. Otherwise, the return value is the correct indicator.
A simple DMA cycle (cont.)

```c
pci_dma_sync_single_for_cpu(pdev, privdata->dma_mem,
    PAGE_SIZE * (1 << memorder),
    PCI_DMA_FROMDEVICE);

for (i=0; i<16; i++)
    printk(KERN_INFO "foo: Read at index %d: %x\n",
        i, privdata->mem[i]);

pci_dma_sync_single_for_device(pdev, privdata->dma_mem,
    PAGE_SIZE * (1 << memorder),
    PCI_DMA_FROMDEVICE);
```
DMA flavors

- Consistent DMA
- “Streaming” DMA mapped and unmapped every use
- “Streaming” DMA mapped once (as shown here)
- Scatter-gather
- See DMA-API-HOWTO.txt in the kernel docs
Letting go

(trivial memory releases excluded)

```c
pci_unmap_single(pdev, privdata->dma_mem,
    PAGE_SIZE * (1 << memorder),
    PCI_DMA_FROMDEVICE);

free_pages ((unsigned long) privdata->mem, memorder);
free_irq(pdev->irq, privdata);
pci_disable_msi(pdev);
pci_clear_master(pdev); /* Nobody seems to do this */
pci_iounmap(pdev, privdata->registers);
pci_release_regions(pdev);
pci_disable_device(pdev);
```
Questions?