How To Write Linux PCI Drivers

Authors: Martin Mares <mj@ucw.cz>
Grant Grundler <grundler@parisc-linux.org>

The world of PCI is vast and full of (mostly unpleasant) surprises. Since each CPU architecture implements different chip-sets and PCI devices have different requirements (erm, "features"), the result is the PCI support in the Linux kernel is not as trivial as one would wish. This short paper tries to introduce all potential driver authors to Linux APIs for PCI device drivers.

A more complete resource is the third edition of "Linux Device Drivers" by Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. LDD3 is available for free (under Creative Commons License) from: https://lwn.net/Kernel/LDD3/.

However, keep in mind that all documents are subject to "bit rot". Refer to the source code if things are not working as described here.

Please send questions/comments/patches about Linux PCI API to the "Linux PCI" <linux-pci@atrey.karlin.mff.cuni.cz> mailing list.

Structure of PCI drivers

PCI drivers "discover" PCI devices in a system via pci_register_driver(). Actually, it's the other way around. When the PCI generic code discovers a new device, the driver with a matching "description" will be notified. Details on this below.

pci_register_driver() leaves most of the probing for devices to the PCI layer and supports online insertion/removal of devices [thus supporting hot-pluggable PCI, CardBus, and Express-Card in a single driver]. pci_register_driver() call requires passing in a table of function pointers and thus dictates the high level structure of a driver.

Once the driver knows about a PCI device and takes ownership, the driver generally needs to perform the following initialization:

- Enable the device
- Request MMIO/IOP resources
- Set the DMA mask size (for both coherent and streaming DMA)
- Allocate and initialize shared control data (pci_allocate_coherent())
- Access device configuration space (if needed)
- Register IRQ handler (request_irq())
- Initialize non-PCI (i.e. LAN/SCSI/etc parts of the chip)
- Enable DMA/processing engines

When done using the device, and perhaps the module needs to be unloaded, the driver needs to take the follow steps:

- Disable the device from generating IRQs
- Release the IRQ (free_irq())
- Stop all DMA activity
- Release DMA buffers (both streaming and coherent)
- Unregister from other subsystems (e.g. scsi or netdev)
- Release MMIO/IOP resources
- Disable the device

Most of these topics are covered in the following sections. For the rest look at LDD3 or <linux/pci.h> .

If the PCI subsystem is not configured (CONFIG_PCI is not set), most of the PCI functions described below are defined as inline functions either completely empty or just returning an appropriate error codes to avoid lots of ifdefs in the drivers.

pci_register_driver() call

PCI device drivers call `pci_register_driver()` during their initialization with a pointer to a structure describing the driver (`struct pci_driver`):

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\PCI\[linux-master][Documentation][PCI]pci.rst`, line 81)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: include/linux/pci.h
   :functions: pci_driver
```

The ID table is an array of `struct pci_device_id` entries ending with an all-zero entry. Definitions with static const are generally preferred.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\PCI\[linux-master] [Documentation] [PCI]pci.rst, line 87)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: include/linux/mod_devicetable.h
   :functions: pci_device_id
```

Most drivers only need `PCI_DEVICE()` or `PCI_DEVICE_CLASS()` to set up a pci_device_id table.

New PCI IDs may be added to a device driver pci_ids table at runtime as shown below:

```
echo "vendor device subvendor subdevice class class_mask driver_data" > \
/sys/bus/pci/drivers/{driver}/new_id
```

All fields are passed in as hexadecimal values (no leading 0x). The vendor and device fields are mandatory, the others are optional. Users need pass only as many optional fields as necessary:

- subvendor and subdevice fields default to PCI_ANY_ID (FFFFFFFF)
- class and classmask fields default to 0
- driver_data defaults to 0UL.
- override_only field defaults to 0.

Note that driver_data must match the value used by any of the pci_device_id entries defined in the driver. This makes the driver_data field mandatory if all the pci_device_id entries have a non-zero driver_data value.

Once added, the driver probe routine will be invoked for any unclaimed PCI devices listed in its (newly updated) pci_ids list.

When the driver exits, it just calls pci_unregister_driver() and the PCI layer automatically calls the remove hook for all devices handled by the driver.

"Attributes" for driver functions/data

Please mark the initialization and cleanup functions where appropriate (the corresponding macros are defined in <linux/init.h>):

__init	Initialization code. Thrown away after the driver initializes.
__exit	Exit code. Ignored for non-modular drivers.

Tips on when/where to use the above attributes:

- The module_init()/module_exit() functions (and all initialization functions called _only_ from these) should be marked __init/__exit.
- Do not mark the struct pci_driver.
- Do NOT mark a function if you are not sure which mark to use. Better to not mark the function than mark the function wrong.

How to find PCI devices manually

PCI drivers should have a really good reason for not using the pci_register_driver() interface to search for PCI devices. The main reason PCI devices are controlled by multiple drivers is because one PCI device implements several different HW services. E.g. combined serial/parallel port/floppy controller.

A manual search may be performed using the following constructs:

Searching by vendor and device ID:

```
struct pci_dev *dev = NULL;
while (dev = pci_get_device(VENDOR_ID, DEVICE_ID, dev))
        configure_device(dev);
```

Searching by class ID (iterate in a similar way):

```
pci_get_class(CLASS_ID, dev)
```

Searching by both vendor/device and subsystem vendor/device ID:

```
pci_get_subsys(VENDOR_ID,DEVICE_ID, SUBSYS_VENDOR_ID, SUBSYS_DEVICE_ID, dev).
```

You can use the constant PCI_ANY_ID as a wildcard replacement for VENDOR_ID or DEVICE_ID. This allows searching for any device from a specific vendor, for example.

These functions are hotplug-safe. They increment the reference count on the pci_dev that they return. You must eventually (possibly at module unload) decrement the reference count on these devices by calling pci_dev_put().

Device Initialization Steps

As noted in the introduction, most PCI drivers need the following steps for device initialization:

- Enable the device
- Request MMIO/IOP resources
- Set the DMA mask size (for both coherent and streaming DMA)
- Allocate and initialize shared control data (pci_allocate_coherent())
- Access device configuration space (if needed)
- Register IRQ handler (request_irq())
- Initialize non-PCI (i.e. LAN/SCSI/etc parts of the chip)
- Enable DMA/processing engines.

The driver can access PCI config space registers at any time. (Well, almost. When running BIST, config space can go away...but that will just result in a PCI Bus Master Abort and config reads will return garbage).

Enable the PCI device

Before touching any device registers, the driver needs to enable the PCI device by calling pci_enable_device(). This will:

- wake up the device if it was in suspended state,
- allocate I/O and memory regions of the device (if BIOS did not),
- allocate an IRQ (if BIOS did not).

Note

pci_enable_device() can fail! Check the return value.

Warning

OS BUG: we don't check resource allocations before enabling those resources. The sequence would make more sense if we called pci_request_resources() before calling pci_enable_device(). Currently, the device drivers can't detect the bug when two devices have been allocated the same range. This is not a common problem and unlikely to get fixed soon.

This has been discussed before but not changed as of 2.6.19: https://lore.kernel.org/r/20060302180025.GC28895@flint.arm.linux.org.uk/

pci_set_master() will enable DMA by setting the bus master bit in the PCI_COMMAND register. It also fixes the latency timer value if it's set to something bogus by the BIOS. pci_clear_master() will disable DMA by clearing the bus master bit.

If the PCI device can use the PCI Memory-Write-Invalidate transaction, call pci_set_mwi(). This enables the PCI_COMMAND bit for Mem-Wr-Inval and also ensures that the cache line size register is set correctly. Check the return value of pci_set_mwi() as not all architectures or chip-sets may support Memory-Write-Invalidate. Alternatively, if Mem-Wr-Inval would be nice to have but is not required, call pci_try_set_mwi() to have the system do its best effort at enabling Mem-Wr-Inval.

Request MMIO/IOP resources

Memory (MMIO), and I/O port addresses should NOT be read directly from the PCI device config space. Use the values in the pci_dev structure as the PCI "bus address" might have been remapped to a "host physical" address by the arch/chip-set specific kernel support.

See Documentation/driver-api/io-mapping.rst for how to access device registers or device memory.

The device driver needs to call pci_request_region() to verify no other device is already using the same address resource. Conversely, drivers should call pci_release_region() AFTER calling pci_disable_device(). The idea is to prevent two devices colliding on the same address range.

Tip

See OS BUG comment above. Currently (2.6.19), The driver can only determine MMIO and IO Port resource availability _after_ calling pci_enable_device().

Generic flavors of pci_request_region() are request_mem_region() (for MMIO ranges) and request_region() (for IO Port ranges). Use these for address resources that are not described by "normal" PCI BARs.

Also see pci_request_selected_regions() below.

Set the DMA mask size

Note

If anything below doesn't make sense, please refer to Documentation/core-api/dma-api.rst. This section is just a reminder that drivers need to indicate DMA capabilities of the device and is not an authoritative source for DMA interfaces.

While all drivers should explicitly indicate the DMA capability (e.g. 32 or 64 bit) of the PCI bus master, devices with more than 32-bit bus master capability for streaming data need the driver to "register" this capability by calling pci_set_dma_mask() with appropriate parameters. In general this allows more efficient DMA on systems where System RAM exists above 4G _physical_ address.

Drivers for all PCI-X and PCIe compliant devices must call set_dma_mask() as they are 64-bit DMA devices.

Similarly, drivers must also "register" this capability if the device can directly address "coherent memory" in System RAM above 4G physical address by calling dma_set_coherent_mask(). Again, this includes drivers for all PCI-X and PCIe compliant devices. Many 64-bit "PCI" devices (before PCI-X) and some PCI-X devices are 64-bit DMA capable for payload ("streaming") data but not control ("coherent") data.

Setup shared control data

Once the DMA masks are set, the driver can allocate "coherent" (a.k.a. shared) memory. See Documentation/core-api/dma-api.rst for a full description of the DMA APIs. This section is just a reminder that it needs to be done before enabling DMA on the device.

Initialize device registers

Some drivers will need specific "capability" fields programmed or other "vendor specific" register initialized or reset. E.g. clearing pending interrupts.

Register IRQ handler

While calling request_irq() is the last step described here, this is often just another intermediate step to initialize a device. This step can often be deferred until the device is opened for use.

All interrupt handlers for IRQ lines should be registered with IRQF_SHARED and use the devid to map IRQs to devices (remember that all PCI IRQ lines can be shared).

request_irq() will associate an interrupt handler and device handle with an interrupt number. Historically interrupt numbers represent IRQ lines which run from the PCI device to the Interrupt controller. With MSI and MSI-X (more below) the interrupt number is a CPU "vector".

request_irq() also enables the interrupt. Make sure the device is quiesced and does not have any interrupts pending before registering the interrupt handler.

MSI and MSI-X are PCI capabilities. Both are "Message Signaled Interrupts" which deliver interrupts to the CPU via a DMA write to a Local APIC. The fundamental difference between MSI and MSI-X is how multiple "vectors" get allocated. MSI requires contiguous blocks of vectors while MSI-X can allocate several individual ones.

MSI capability can be enabled by calling pci_alloc_irq_vectors() with the PCI_IRQ_MSI and/or PCI_IRQ_MSIX flags before calling request_irq(). This causes the PCI support to program CPU vector data into the PCI device capability registers. Many architectures, chip-sets, or BIOSes do NOT support MSI or MSI-X and a call to pci_alloc_irq_vectors with just the PCI_IRQ_MSI and PCI_IRQ_MSIX flags will fail, so try to always specify PCI_IRQ_LEGACY as well.

Drivers that have different interrupt handlers for MSI/MSI-X and legacy INTx should chose the right one based on the msi_enabled and msix_enabled flags in the pci_dev structure after calling pci_alloc_irq_vectors.

There are (at least) two really good reasons for using MSI:

1. MSI is an exclusive interrupt vector by definition. This means the interrupt handler doesn't have to verify its device caused the interrupt.
2. MSI avoids DMA/IRQ race conditions. DMA to host memory is guaranteed to be visible to the host CPU(s) when the MSI is delivered. This is important for both data coherency and avoiding stale control data. This guarantee allows the driver to omit MMIO reads to flush the DMA stream.

See drivers/infiniband/hw/mthca/ or drivers/net/tg3.c for examples of MSI/MSI-X usage.

PCI device shutdown

When a PCI device driver is being unloaded, most of the following steps need to be performed:

- Disable the device from generating IRQs
- Release the IRQ (free_irq())
- Stop all DMA activity
- Release DMA buffers (both streaming and coherent)

- Unregister from other subsystems (e.g. scsi or netdev)
- Disable device from responding to MMIO/IO Port addresses
- Release MMIO/IO Port resource(s)

Stop IRQs on the device

How to do this is chip/device specific. If it's not done, it opens the possibility of a "screaming interrupt" if (and only if) the IRQ is shared with another device.

When the shared IRQ handler is "unhooked", the remaining devices using the same IRQ line will still need the IRQ enabled. Thus if the "unhooked" device asserts IRQ line, the system will respond assuming it was one of the remaining devices asserted the IRQ line. Since none of the other devices will handle the IRQ, the system will "hang" until it decides the IRQ isn't going to get handled and masks the IRQ (100,000 iterations later). Once the shared IRQ is masked, the remaining devices will stop functioning properly. Not a nice situation.

This is another reason to use MSI or MSI-X if it's available. MSI and MSI-X are defined to be exclusive interrupts and thus are not susceptible to the "screaming interrupt" problem.

Release the IRQ

Once the device is quiesced (no more IRQs), one can call free_irq(). This function will return control once any pending IRQs are handled, "unhook" the drivers IRQ handler from that IRQ, and finally release the IRQ if no one else is using it.

Stop all DMA activity

It's extremely important to stop all DMA operations BEFORE attempting to deallocate DMA control data. Failure to do so can result in memory corruption, hangs, and on some chip-sets a hard crash.

Stopping DMA after stopping the IRQs can avoid races where the IRQ handler might restart DMA engines.

While this step sounds obvious and trivial, several "mature" drivers didn't get this step right in the past.

Release DMA buffers

Once DMA is stopped, clean up streaming DMA first. I.e. unmap data buffers and return buffers to "upstream" owners if there is one.

Then clean up "coherent" buffers which contain the control data.

See Documentation/core-api/dma-api.rst for details on unmapping interfaces.

Unregister from other subsystems

Most low level PCI device drivers support some other subsystem like USB, ALSA, SCSI, NetDev, Infiniband, etc. Make sure your driver isn't losing resources from that other subsystem. If this happens, typically the symptom is an Oops (panic) when the subsystem attempts to call into a driver that has been unloaded.

Disable Device from responding to MMIO/IO Port addresses

io_unmap() MMIO or IO Port resources and then call pci_disable_device(). This is the symmetric opposite of pci_enable_device(). Do not access device registers after calling pci_disable_device().

Release MMIO/IO Port Resource(s)

Call pci_release_region() to mark the MMIO or IO Port range as available. Failure to do so usually results in the inability to reload the driver.

How to access PCI config space

You can use *pci_(read|write)_config_(byte|word|dword)* to access the config space of a device represented by *struct pci_dev **. All these functions return 0 when successful or an error code (*PCIBIOS_...*) which can be translated to a text string by pcibios_strerror. Most drivers expect that accesses to valid PCI devices don't fail.

If you don't have a struct pci_dev available, you can call *pci_bus_(read|write)_config_(byte|word|dword)* to access a given device and function on that bus.

If you access fields in the standard portion of the config header, please use symbolic names of locations and bits declared in <linux/pci.h>.

If you need to access Extended PCI Capability registers, just call pci_find_capability() for the particular capability and it will find the corresponding register block for you.

Other interesting functions

pci_get_domain_bus_and_slot()	Find pci_dev corresponding to given domain, bus and slot and number. If the device is found, its reference count is increased.
pci_set_power_state()	Set PCI Power Management state (0=D0 ... 3=D3)
pci_find_capability()	Find specified capability in device's capability list.
pci_resource_start()	Returns bus start address for a given PCI region
pci_resource_end()	Returns bus end address for a given PCI region
pci_resource_len()	Returns the byte length of a PCI region
pci_set_drvdata()	Set private driver data pointer for a pci_dev
pci_get_drvdata()	Return private driver data pointer for a pci_dev
pci_set_mwi()	Enable Memory-Write-Invalidate transactions.
pci_clear_mwi()	Disable Memory-Write-Invalidate transactions.

Miscellaneous hints

When displaying PCI device names to the user (for example when a driver wants to tell the user what card has it found), please use pci_name(pci_dev).

Always refer to the PCI devices by a pointer to the pci_dev structure. All PCI layer functions use this identification and it's the only reasonable one. Don't use bus/slot/function numbers except for very special purposes -- on systems with multiple primary buses their semantics can be pretty complex.

Don't try to turn on Fast Back to Back writes in your driver. All devices on the bus need to be capable of doing it, so this is something which needs to be handled by platform and generic code, not individual drivers.

Vendor and device identifications

Do not add new device or vendor IDs to include/linux/pci_ids.h unless they are shared across multiple drivers. You can add private definitions in your driver if they're helpful, or just use plain hex constants.

The device IDs are arbitrary hex numbers (vendor controlled) and normally used only in a single location, the pci_device_id table.

Please DO submit new vendor/device IDs to https://pci-ids.ucw.cz/. There's a mirror of the pci.ids file at https://github.com/pciutils/pciids.

Obsolete functions

There are several functions which you might come across when trying to port an old driver to the new PCI interface. They are no longer present in the kernel as they aren't compatible with hotplug or PCI domains or having sane locking.

pci_find_device()	Superseded by pci_get_device()
pci_find_subsys()	Superseded by pci_get_subsys()
pci_find_slot()	Superseded by pci_get_domain_bus_and_slot()
pci_get_slot()	Superseded by pci_get_domain_bus_and_slot()

The alternative is the traditional PCI device driver that walks PCI device lists. This is still possible but discouraged.

MMIO Space and "Write Posting"

Converting a driver from using I/O Port space to using MMIO space often requires some additional changes. Specifically, "write posting" needs to be handled. Many drivers (e.g. tg3, acenic, sym53c8xx_2) already do this. I/O Port space guarantees write transactions reach the PCI device before the CPU can continue. Writes to MMIO space allow the CPU to continue before the transaction reaches the PCI device. HW weenies call this "Write Posting" because the write completion is "posted" to the CPU before the transaction has reached its destination.

Thus, timing sensitive code should add readl() where the CPU is expected to wait before doing other work. The classic "bit banging" sequence works fine for I/O Port space:

```
for (i = 8; --i; val >>= 1) {
        outb(val & 1, ioport_reg);      /* write bit */
        udelay(10);
}
```

The same sequence for MMIO space should be:

```
for (i = 8; --i; val >>= 1) {
        writeb(val & 1, mmio_reg);      /* write bit */
        readb(safe_mmio_reg);           /* flush posted write */
        udelay(10);
```

```
    }
```

It is important that "safe_mmio_reg" not have any side effects that interferes with the correct operation of the device.

Another case to watch out for is when resetting a PCI device. Use PCI Configuration space reads to flush the writel(). This will gracefully handle the PCI master abort on all platforms if the PCI device is expected to not respond to a readl(). Most x86 platforms will allow MMIO reads to master abort (a.k.a. "Soft Fail") and return garbage (e.g. ~0). But many RISC platforms will crash (a.k.a."Hard Fail").