Device drivers

Michal Schmidt
Principal Software Engineer
Network Drivers team

Red Hat

Overview

- Role of a device driver
- Drivers as modules
- Device model
- Character devices
- Block devices
- Device discovery
- Device I/O
- Interrupt handling
- Direct memory access (DMA)
- Network drivers

Role of a device driver

- Provide portability.
  - Abstract away most differences between devices.
- Communicates with the device it controls.
- Interacts with the relevant kernel subsystem.
  - Calls kernel API functions.
  - Provides callbacks for a kernel subsystem to call.

Device drivers in the kernel tree

- `sloccount .   # v6.12`

```
[...]
SLOC    Directory  SLOC-by-Language (Sorted)
18548831 drivers      ansic=18539246,asm=4567,yacc=1679,python=1485,perl=792,
                      lex=771,sh=291
1782803 arch          ansic=1509082,asm=259426,perl=12102,sh=1426,awk=762,
                      sed=5
1205419 sound         ansic=1205419
1166669 fs            ansic=1166669
1001314 tools         ansic=813911,sh=122122,python=51806,perl=4881,asm=4672,
                      yacc=1707,cpp=1141,lex=991,awk=58,ruby=25
943556  net           ansic=943556
795685  include       ansic=793238,cpp=2447
319247  kernel        ansic=319130,asm=60,sh=57
182401  lib           ansic=182225,perl=123,sh=34,awk=13,asm=6
126769  mm            ansic=126769
[...]
```

Device drivers in the kernel tree

- sloccount drivers/ # v6.12

```
SLOC  DirectorySLOC-by-Language (Sorted)
6260564 gpu           ansic=6256468,asm=2849,python=956,sh=291
3733687 net           ansic=3732998,asm=689
1098377 media         ansic=1098377
730815  scsi          ansic=727491,yacc=1679,lex=771,perl=762,asm=112
569058  clk           ansic=569058
441900  pinctrl       ansic=441900
409954  usb           ansic=409924,perl=30
340275  infiniband    ansic=340275
319974  staging       ansic=319974
264168  iio           ansic=264168
221691  video         ansic=221691
199652  crypto        ansic=199594,asm=58
156994  input         ansic=156994
```

Device drivers as a part of the kernel

- Linux is a monolithic kernel.
  - Driver code runs at the same protection level and with the same address space as the core kernel.
- Privileged code. Bugs can crash the system.
- No protection from resource (memory, ...) leaks.
- A module can call functions that are exported by the kernel.
  - *EXPORT_SYMBOL(_printk);*
  - *EXPORT_SYMBOL_GPL(synchronize_rcu);*

Red Hat

Hello World Module

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");
static int hello_init(void)
{
   printk(KERN_ALERT "Hello, world\n");
   return 0;
}
static void hello_exit(void)
{
   printk(KERN_ALERT "Goodbye, world\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

Building a module

- `make`
- A simple *Makefile* for an in-tree module could be just:

```
obj-$(CONFIG_HELLO) += hello.o
```

- Where *CONFIG_HELLO* would be defined in a *Kconfig* file:

```
config HELLO
    tristate “Hello World”
    help
        This is a Hello World module.
```

- For info about out-of-tree (external) builds, see https://docs.kernel.org/kbuild/modules.html

Loading / unloading modules

- `insmod ./hello.ko` - load a module from a file.
- `modprobe hello` - load an installed module, with dependencies.
- `rmmod hello` - unload a module.
- `lsmod` - list loaded modules.
- `modinfo hello` - show information about a module.
- Modules can take options
  - on the *insmod/modprobe* command line
  - */etc/modprobe.d/*
  - on the kernel command line

Good to know

- Version dependency
  - In-kernel APIs are not frozen, they evolve with each version.
  - Driver written for one kernel version may not compile or run correctly on a different kernel version.
- Platform dependency
  - Driver code is mostly CPU arch independent.
  - It needs to use APIs for endian conversions, memory barriers, … correctly.
- License
  - Loading a non-GPL licensed module taints the kernel.

Red Hat

Kernel device model

- Devices are attached to buses and organized in a tree-like structure.
- Necessary for device discovery, power management.
- The *sysfs* virtual filesystem, mounted on */sys*, presents the model to userspace.
- Internally, every node is represented with:

  ```
  struct device dev;
  ```

  Typically embedded in a larger struct (*pci_dev*, *pci_bus*, *input_dev*, *usb_device*, *gnss_device, …*).
- A device belongs to a device *class* (“net”, “block”, “pci_bus”, “input”, “sound”, “tty”, “gnss”, …).
- Devices of the same class expose similar functionality and sysfs attributes.
- Device objects are refcounted: *get_device()*, *put_device()*
- They have a destructor, *dev.release*, called when the refcount drops to zero.
- https://docs.kernel.org/driver-api/driver-model/index.html

Registering a device class

- A device *class* can be defined as a static instance (example from *net/core/net-sysfs.c*):

```
static const struct class net_class = {
    .name = "net",
    .dev_release = netdev_release,    // destructor for devices of this class
    .dev_groups = net_class_groups,  // groups of sysfs attributes
                    // These groups are defined with a bit of macro magic. Look for
                    //  ATTRIBUTE_GROUPS(net_class);
                    // and uses of the DEVICE_ATTR_{RO,RW} macros.
/* … */ };
```

- Such a class is registered with *class_register()*:

```
err = class_register(&net_class);
```

- … and unregistered with *class_unregister()*.

Red Hat

Dynamic creation of a device class

- Alternatively, a class can be created dynamically (example from *drivers/gnss/core.c*):

```
static struct class *gnss_class;
```

In *gnss_module_init():*

```
    gnss_class = class_create("gnss");
    if (IS_ERR(gnss_class)) { /* … */ }
    gnss_class->dev_groups = gnss_groups;
```

- *class_create()* already calls *class_register()*.
- Undo with *class_destroy()*:

```
class_destroy(gnss_class);
```

Types of devices

- Character device
  - Stream of bytes
  - */dev/null*, */dev/random*, */dev/tty1*, ...
- Block device
  - I/O operations on 1 or more blocks
  - The block layer works with 512 bytes blocks.
  - */dev/sda*, */dev/nvme0n1*, ...
- Network device
  - Sending and receiving packets
  - No */dev* nodes
  - Most applications don't open them directly, but use them through the *socket* interface.

Character device

Major and minor device numbers

- `ls -l /dev/zero`

```
crw-rw-rw-. 1 root root 1, 5 Nov 22 09:03 /dev/zero
```

This is a **c**haracter device
(**b** for block devices)

Major number

Minor number

- Major - identifies the driver (traditionally; sharing is possible)
- Minor - a specific device of the driver
- Common device types have their major:minor numbers statically assigned.
- Dynamic allocation for the rest
- Network devices are not represented as special files and don’t have a major:minor.

Device numbers in the kernel

- The complete device number is represented as an integer type

```
typedef … dev_t;
```

- Macros to extract the major and minor numbers:

```
unsigned int major = MAJOR(my_dev_t);
unsigned int minor = MINOR(my_dev_t);
```

- To compose a *dev_t* value from the two numbers:

```
dev_t my_dev_t = MKDEV(major, minor);
```

Allocating device numbers

- Use *alloc_chrdev_region()* to allocate a range of char device numbers dynamically.
- Example from *drivers/gnss/core.c:gnss_module_init()*:

  ```
  ret = alloc_chrdev_region(&gnss_first, 0, GNSS_MINORS, "gnss");
  ```

  gnss_first (a variable of type *dev_t*) will be set to the first device number of the allocated range.
- Avoid in new code: *register_chrdev_region()* - for statically assigned device number ranges.

  Example from *drivers/input/input.c:input_init()*:

  ```
  err = register_chrdev_region(MKDEV(INPUT_MAJOR, 0),
                               INPUT_MAX_CHAR_DEVICES, "input");
  ```
- Dynamic or static, free them with *unregister_chrdev_region()*:

  ```
  unregister_chrdev_region(gnss_first, GNSS_MINORS);
  ```
- See currently allocated major numbers in */proc/devices*:

  ```
  # modprobe gnss && grep -E '(gnss|input)' /proc/devices
   13 input
  511 gnss
  ```

Character device example

- *drivers/input/mousedev.c* - Implementation of */dev/input/{mouse*N*,mice}* devices.
- A mouse device is represented as:

```
struct mousedev {
    /* … */
    struct device dev;   // for the device model
    struct cdev cdev;    // a character device
    /* … */
};
```

Character device creation

- *drivers/input/mousedev.c:mousedev_create()* allocates and registers a mouse device:

```
struct mousedev *mousedev;
mousedev = kzalloc(sizeof(struct mousedev), GFP_KERNEL);
dev_set_name(&mousedev->dev, "mouse%d", minor);
mousedev->dev.class = &input_class;          // class registered in drivers/input/input.c
mousedev->dev.parent = &input_dev->dev;      // parent device in the sysfs hierarchy
mousedev->dev.devt = MKDEV(INPUT_MAJOR, minor);  // device number
mousedev->dev.release = mousedev_free;          // destructor
device_initialize(&mousedev->dev);              // internal dev fields
cdev_init(&mousedev->cdev, &mousedev_fops);     // internal cdev fields + assign file ops
err = cdev_device_add(&mousedev->cdev, &mousedev->dev);
```

- The character device now appears in */sys* and */dev* and can be opened from userspace.

File operations

- ```
  static ssize_t mousedev_read(struct file *file, char __user *buffer,
                               size_t count, loff_t *ppos);
  …
  ```
- ```
  static const struct file_operations mousedev_fops = {
      .owner      = THIS_MODULE,        // prevents in-use module unload
      .read       = mousedev_read,      // read syscall
      .write      = mousedev_write,     // write syscall
      .poll       = mousedev_poll,      // select/poll/epoll
      .open       = mousedev_open,      // open syscall
      .release    = mousedev_release,   // last close
      .fasync     = mousedev_fasync,    // for O_ASYNC (man fcntl)
      .llseek     = noop_llseek,        // lseek syscall will do nothing, with success
  };
  ```
- There are many more ops: *.mmap*, *.fsync*, *.flush*, *.fallocate*, *unlocked_ioctl*, …

Block device

Block Driver

- Register a block device with:
  int ***register_blkdev****(unsigned int major, const char *name);*
  - If *major* is 0, a new one will be allocated and returned.
- Describe a "tag set" with *struct blk_mq_tag_set*, that has a pointer to *struct blk_mq_ops*.
  - ***.queue_rq*** – queue a new request for block I/O.
  - ***.complete*** – mark a request as complete.
- Allocate the tag set with:
  int **blk_mq_alloc_tag_set***(struct blk_mq_tag_set *set);*
- Allocate a *gendisk* with:
  *struct gendisk **blk_mq_alloc_disk***(struct blk_mq_tag_set *set, struct queue_limits *lim, void *queuedata);*
- Fill the *struct gendisk* members, including *fops*, a pointer to *struct block_device_operations*.
  - *.open*, *.release*, *.ioctl*, …
- Add the disk with:
  int ***add_disk****(struct gendisk *disk);*

Block device example

- For an example, see the loop device driver
  - drivers/block/loop.c
  - https://man7.org/linux/man-pages/man4/loop.4.html
- An even simpler block driver is the ramdisk
  - drivers/block/brd.c
  - single-queue

Driver in examples: 8139cp

- In the next sections, we'll use the *8139cp* driver in examples.
- Old device driver, added in Linux v2.4.13 (2001).
- RealTek RTL-8139 C+ PCI Fast Ethernet Adapter
  - "Fast Ethernet" means 100 Mbit/s.
- PCI driver. Network driver.
- Small, easy to understand, all in one source file
  - [*drivers/net/ethernet/realtek/8139cp.c*] has ~2000 lines.
- QEMU can emulate the hardware for virtual machines.
  - *-netdev user,id=mynet -device rtl8139,netdev=mynet*
  - libvirt (virt-manager, ...): *<model type="rtl8139"/>*

Device discovery

PCI identifiers (vendor:device)

```
$ lspci -nnv -s 01:00.0
01:00.0 Ethernet controller [0200]: Realtek Semiconductor Co., Ltd. RTL-8100/8101L/8139
        PCI Fast Ethernet Adapter [10ec:8139] (rev 20)
    Subsystem: Red Hat, Inc. QEMU Virtual Machine [1af4:1100]
    …
```

Module aliases

```
$ cat /sys/devices/pci0000:00/0000:00:02.0/0000:01:00.0/modalias
pci:v000010ECd00008139sv00001AF4sd00001100bc02sc00i00

$ modinfo 8139cp
filename:    /lib/modules/[…]/kernel/drivers/net/ethernet/realtek/8139cp.ko.xz
license:     GPL
version:     1.3
description: RealTek RTL-8139C+ series 10/100 PCI Ethernet driver
author:      Jeff Garzik <jgarzik@pobox.com>
srcversion:  FC67FEB72AC9581B21B24CF
alias:       pci:v00000357d0000000Asv*sd*bc*sc*i*
alias:       pci:v000010ECd00008139sv*sd*bc*sc*i*
depends:     mii
…
```

Declaring supported device in a module

```
static const struct pci_device_id cp_pci_tbl[] = {
    { PCI_DEVICE(PCI_VENDOR_ID_REALTEK, PCI_DEVICE_ID_REALTEK_8139), },
    { PCI_DEVICE(PCI_VENDOR_ID_TTTECH,  PCI_DEVICE_ID_TTTECH_MC322), },
    { },
};

MODULE_DEVICE_TABLE(pci, cp_pci_tbl);
```

pci_device_id

- Describes the types of PCI devices this driver supports

```
struct pci_device_id {
   __u32 vendor, device;
   __u32 subvendor, subdevice;
   __u32 class, class_mask;
   kernel_ulong_t driver_data;
};
```

Registering a PCI driver

- Register and unregister your PCI driver in the module init and exit functions.

```
static int __init pci_skel_init(void)
{
   return pci_register_driver(&pci_driver);
}
static void __exit pci_skel_exit(void)
{
   pci_unregister_driver(&pci_driver);
}
```

- To create trivial init/exit functions like the above, a simple macro can do it for you:

```
module_pci_driver(cp_driver);
```

Describing a PCI driver

- *struct pci_driver* is the structure needed to register a PCI driver with the kernel

```
static struct pci_driver cp_driver = {
    .name          = DRV_NAME,
    .id_table      = cp_pci_tbl,
    .probe         = cp_init_one,
    .remove        = cp_remove_one,
    .driver.pm     = &cp_pm_ops,
};
```

Probing a PCI device

- *int (*probe)(struct pci_dev *dev, const struct pci_device_id *id);*
  - Called when the kernel detects a device belonging to the driver.
  - *dev* points to the PCI device being probed.
  - *id* points to the matching entry from the driver's device table.
- `.probe = cp_init_one`
-
```
static int cp_init_one (struct pci_dev *pdev, const struct pci_device_id *ent)
{
    /* … */
    if (pdev->vendor == PCI_VENDOR_ID_REALTEK &&
         pdev->device == PCI_DEVICE_ID_REALTEK_8139 && pdev->revision < 0x20) {
        /* … error, incompatible device */
        return -ENODEV;
    }
    /* … */
    rc = pci_enable_device(pdev);
    /* … */
    return 0;
}
```

Removing a PCI device

- *void (*remove)(struct pci_dev *dev);*
  - Called when the kernel unbinds the driver from the device. E.g.
    - *rmmod 8139cp*
    - *echo 0000:01:00.0 > /sys/bus/pci/drivers/8139cp/unbind*
  - *dev* points to the PCI device being removed.
- `.remove = cp_remove_one`
- `static void cp_remove_one(struct pci_dev *pdev)`

```
{
    /* … */
    pci_disable_device(pdev);
    /* … */
}
```

Device I/O

The driver has to be able to read and write from/to devices' registers.

- Port I/O
- Memory-mapped I/O

Port I/O

- Mostly legacy only
- x86 has a 16-bit I/O port space, accessed using *in*, *out* instructions.
- Writing a character to serial port (COM1):

```
movw $0x3F8, %dx   # Load I/O port address 0x3F8 into DX
movb $0x41, %al    # Load ASCII value for 'A' (0x41) into AL
outb %al, %dx      # Write the byte from AL to the port in DX (0x3F8)
```

- From C code in the kernel:

```
outb('A', 0x3f8);
```

- Write a byte/word(2B)/long(4B): *outb(), outw(), outl()*
- Read a byte/word(2B)/long(4B): *inb(), inw(), inl()*

Memory-mapped I/O (MMIO)

- Used for most devices. Registers are represented as a memory range.
- Available memory address space is much larger than the 16-bit port space.
- See the I/O port and memory ranges of a PCI device:

```
# lspci -v -s 01:00.0
01:00.0 Ethernet controller: Realtek Semiconductor Co., Ltd. RTL-8100/8101L/8139 PCI Fast Ethernet
Adapter (rev 20)
	Subsystem: Red Hat, Inc. QEMU Virtual Machine
	Physical Slot: 0
	Control: I/O+ Mem+ BusMaster+ SpecCycle- MemWINV- VGASnoop- ParErr- Stepping- SERR+ FastB2B-
	DisINTx-
	Status: Cap- 66MHz- UDF- FastB2B- ParErr- DEVSEL=fast >TAbort- <TAbort- <MAbort- >SERR- <PERR- INTx-
	Latency: 0, Cache Line Size: 64 bytes
	Interrupt: pin A routed to IRQ 22
	Region 0: I/O ports at c000 [size=256]
	Region 1: Memory at fda40000 (32-bit, non-prefetchable) [size=256]
	Expansion ROM at fda00000 [disabled] [size=256K]
	Kernel driver in use: 8139cp
	Kernel modules: 8139cp, 8139too
```

MMIO in the driver

- How the *8139cp* driver finds and maps the memory region (simplified):

```
#define DRV_NAME        "8139cp"
#define CP_REGS_SIZE 256
static int cp_init_one (struct pci_dev *pdev, const struct pci_device_id *ent)
{
    struct cp_private *cp;
    /* … */
    rc = pci_request_regions(pdev, DRV_NAME);
    if (rc) { /* …error… */ }
    pciaddr = pci_resource_start(pdev, 1);
    if (!pciaddr) { /* …error… */ }
    if (pci_resource_len(pdev, 1) < CP_REGS_SIZE) { /* …error… */ }
    regs = ioremap(pciaddr, CP_REGS_SIZE);
    if (!regs) { /* …error… */ }
    cp->regs = regs;
    /* … */
}
```

- The reserved regions can be seen in */proc/iomem*, */proc/ioports*.
- When removing the driver, undo the allocations. In *cp_remove_one()*:

```
iounmap(cp->regs);
pci_release_regions(pdev);
```


Red Hat

Devres: managed device resources

- There are convenient wrappers to help with releasing resources correctly.
- The code on the previous slide could be simplified by using *pcim_iomap_region()*.
- No need to write code to release the resource in the driver's *.remove()* callback.
- Error paths are simplified.
- For all the device managed APIs (*devm_**, *pcim_**), see https://docs.kernel.org/driver-api/driver-model/devres.html

MMIO reading and writing

- To read from and write to the memory-mapped region,
  use (for 8, 16, 32, 64 bit access, respectively):
  readb(), readw(), readl(), readq()
  writeb(), writew(), writel(), writeq()
- *8139cp* uses its own wrapper macros to read and write to the device registers:

```
#define cpr16(reg)       readw(cp->regs + (reg))
#define cpw16(reg,val)   writew((val), cp->regs + (reg))
```

  ...
- PCI writes are posted asynchronously!
  If a driver needs to ensure a write has made it to the device, it has to issue a read from the same device.

Red Hat

Interrupts

The device needs to be able to raise the driver’s attention when

- an operation finishes,
- new data becomes available,
- status changes,
- an error condition appears.

Interrupt handler

```
static irqreturn_t cp_interrupt (int irq,
                                 void *dev_instance)
{
    struct net_device *dev = dev_instance;
    struct cp_private *cp = netdev_priv(dev);
    int handled = 0;
    u16 status;

    // Read the interrupt status register
    // to check if the interrupt was raised
    // by our device.
    status = cpr16(IntrStatus);
    if (!status || (status == 0xFFFF))
            goto out;

    // OK, the interrupt is for us
    handled = 1;

    // Tell the device we got the interrupt request.
    // The device can shut up about it now.
    cpw16(IntrStatus, /* … */);

    /* …Perform appropriate actions… */

out:
    // Return one of these possible values:
    //  IRQ_NONE (0) - I did not handle
    //               the interrupt. It was not mine.
    //  IRQ_HANDLED (1) - I handled the interrupt.
    //  IRQ_WAKE_THREAD (2) - The interrupt is for me
    //           and I want to handle it in a thread.
    return IRQ_RETVAL(handled);
}
```


Red Hat

Interrupt handler registration

- The driver has to register the interrupt handler with the kernel.

```
typedef irqreturn_t (*irq_handler_t)(int, void *);

/**
 * request_irq - Add a handler for an interrupt line
 * @irq:        The interrupt line to allocate
 * @handler:    Function to be called when the IRQ occurs.
 * @flags:      Handling flags
 * @name:       Name of the device generating this interrupt
 * @dev:        A cookie passed to the handler function
 … */
static inline int __must_check
request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags,
            const char *name, void *dev)
{ /*…*/ }
```

- Most commonly used values for `flags`:

`IRQF_SHARED` - The IRQ line may be shared between 2 or more devices.

`0` - The IRQ is not shared. Typical for modern MSI-X interrupts.

Interrupt handler registration example

- An interrupt may fire immediately after registering the handler. Be prepared!

```
static int cp_open (struct net_device *dev)
{
    struct cp_private *cp = netdev_priv(dev);
    const int irq = cp->pdev->irq;
    int rc;
    // Initializations first!
    rc = cp_alloc_rings(cp);
    if (rc)
        return rc;
    napi_enable(&cp->napi);
    cp_init_hw(cp);
    rc = request_irq(irq, cp_interrupt, IRQF_SHARED, dev->name, dev);
    /* … */
}
```

- See registered interrupt handlers in */proc/interrupts*:

```
$ grep enp1s0 /proc/interrupts
22:     101   IO-APIC  22-fasteoi   virtio3, virtio2, enp1s0
```

Interrupt context quiz

Things you can or cannot do in an interrupt handler.

- Read and write device registers? Yes
- `mutex_lock(...)` No
- `spin_lock(...)` Yes
- `schedule_work(...)` Yes
- `kmalloc(sizeof(my_data), GFP_KERNEL)` No
- `kmalloc(sizeof(my_data), GFP_ATOMIC)` Yes

In an interrupt handler, what does *current* point to?

Freeing the interrupt handler

- The inverse of *request_irq()* is:

```
/**
 *      free_irq - free an interrupt allocated with request_irq
 *      @irq: Interrupt line to free
 *      @dev_id: Device identity to free
 *
 *      Remove an interrupt handler. The handler is removed and if the
 *      interrupt line is no longer in use by any driver it is disabled.
 *      On a shared IRQ the caller must ensure the interrupt is disabled
 *      on the card it drives before calling this function. The function
 *      does not return until any executing interrupts for this IRQ
 *      have completed.
 *
 *      This function must not be called from interrupt context.
 *
 *      Returns the devname argument passed to request_irq.
 */
const void *free_irq(unsigned int irq, void *dev_id);
```

Threaded interrupt handlers

- If handling the interrupt requires doing some of the things that are not allowed in interrupt context, one of the options is to use a threaded interrupt handler.

```
int request_threaded_irq(unsigned int irq, irq_handler_t handler,
                         irq_handler_t thread_fn, unsigned long irqflags,
                         const char *devname, void *dev_id);
```

- Like `request_irq()`, but you provide **2** *irq_handler_t* functions.
- *handler* runs in interrupt context. If it returns *IRQ_WAKE_THREAD*, *thread_fn* will be run, in process context.

Red Hat

Legacy PCI interrupts ("INTx")

- For PCI devices, *struct pci_dev* has the *irq* member, telling the driver the IRQ number it can use.
- In *cp_open()*, we saw the driver uses *cp->**pdev->irq***.
- This is the PCI device's **legacy interrupt** number.
- A classic PCI card has 4 IRQ pins (marked A, B, C, D) that the PCI bus physically connects to the interrupt controller. The card (PCI function) uses one of these pins to signal its interrupt requests. A register (*PCI_INTERRUPT_PIN*) in the card's PCI configuration space tells us which pin it uses. In `lspci` we saw:

  ```
  Interrupt: pin A routed to IRQ 22
  ```
- The IRQ number is assigned by the BIOS and written back to the device into another register (*PCI_INTERRUPT_LINE*) in the PCI configuration space. The kernel may also rewrite it as it does its own IRQ routing. Drivers do not have to care.
- https://en.wikipedia.org/wiki/PCI_configuration_space

Message-signalled interrupts (MSI, MSI-X)

- Modern PCIe devices can signal interrupts using an in-band message, instead of dedicated physical IRQ lines.
- Advantages:
  - The interrupt signal is ordered with data messages. MSI-X message will not arrive to the CPU before a previously sent DMA write from the device.
  - Many more possible interrupts. With MSI-X, a device can allocate up to 2048 interrupts.
  - No need to share IRQs between drivers.
- Why would a device need more than one IRQ?
  - Performance
  - Multi-queue network adapters, NVME storage.
  - Each device queue can have its own IRQ, routed to a different CPU.

MSI-X in action

- The old *8139cp* cannot do MSI(-X). Here's *iwlwifi* wireless driver:

```
$ awk '/iwlwifi/{print $1,$(NF-2),$(NF-1),$NF;}' /proc/interrupts
186: IR-PCI-MSIX-0000:09:00.0 0-edge iwlwifi:default_queue
187: IR-PCI-MSIX-0000:09:00.0 1-edge iwlwifi:queue_1
188: IR-PCI-MSIX-0000:09:00.0 2-edge iwlwifi:queue_2
189: IR-PCI-MSIX-0000:09:00.0 3-edge iwlwifi:queue_3
190: IR-PCI-MSIX-0000:09:00.0 4-edge iwlwifi:queue_4
191: IR-PCI-MSIX-0000:09:00.0 5-edge iwlwifi:queue_5
192: IR-PCI-MSIX-0000:09:00.0 6-edge iwlwifi:queue_6
193: IR-PCI-MSIX-0000:09:00.0 7-edge iwlwifi:queue_7
194: IR-PCI-MSIX-0000:09:00.0 8-edge iwlwifi:queue_8
195: IR-PCI-MSIX-0000:09:00.0 9-edge iwlwifi:queue_9
196: IR-PCI-MSIX-0000:09:00.0 10-edge iwlwifi:queue_10
197: IR-PCI-MSIX-0000:09:00.0 11-edge iwlwifi:queue_11
198: IR-PCI-MSIX-0000:09:00.0 12-edge iwlwifi:queue_12
199: IR-PCI-MSIX-0000:09:00.0 13-edge iwlwifi:queue_13
200: IR-PCI-MSIX-0000:09:00.0 14-edge iwlwifi:queue_14
201: IR-PCI-MSIX-0000:09:00.0 15-edge iwlwifi:exception
```

Obtaining MSI-X in a driver

- *iwlwifi* uses old MSI-X API:

```
/**
 * pci_enable_msix_range() - Enable MSI-X interrupt mode on device
 * @dev:    the PCI device to operate on
 * @entries: input/output parameter, array of MSI-X configuration entries
 * @minvec:  minimum required number of MSI-X vectors
 * @maxvec:  maximum desired number of MSI-X vectors
 …
 */
int pci_enable_msix_range(struct pci_dev *dev, struct msix_entry *entries,
                          int minvec, int maxvec);
```

- When successful, *entries[i].vector* contains an IRQ number to use with *request_irq()* or *request_threaded_irq()*.
- Undo with:

```
void pci_disable_msix(struct pci_dev *dev);
```


Red Hat

Newer MSI-X API

- *pci_enable_msix_range()* is already deprecated in favor of:

```
/**
 * pci_alloc_irq_vectors() - Allocate multiple device interrupt vectors
 * @dev:    the PCI device to operate on
 * @min_vecs: minimum required number of vectors (must be >= 1)
 * @max_vecs: maximum desired number of vectors
 * @flags: One or more of: [...]
 [...]
 * Upon a successful allocation, the caller should use pci_irq_vector()
 * to get the Linux IRQ number to be passed to request_threaded_irq().
 * The driver must call pci_free_irq_vectors() on cleanup.
 *
 * Return: number of allocated vectors [... or -errno]
 */
int pci_alloc_irq_vectors(struct pci_dev *dev, unsigned int min_vecs,
                      unsigned int max_vecs, unsigned int flags);

int pci_irq_vector(struct pci_dev *dev, unsigned int nr);
```

- https://docs.kernel.org/PCI/msi-howto.html

Direct Memory Access (DMA)

- Data transfers without involving the CPU.
- Several components need to work together
  - the device itself
  - IOMMU – I/O memory management unit
    - Maps bus addresses to physical memory addresses.
    - Provides protection from misbehaving devices.
  - PCIe controller
- A device driver is abstracted away from this by using the DMA API.
  - https://docs.kernel.org/core-api/dma-api-howto.html
- Before a DMA transfer, the driver needs to use the API to set up a mapping for a range of memory addresses and bus addresses.

Types of DMA mappings

- “**Consistent**” a.k.a. “**Coherent**”
  - Typically longer-lived, set up at driver initialization / reconfiguration time.
  - The device and the CPU can both access the data in parallel and will see updates made by each other without any explicit software flushing.
  - Typical use in a network driver: for the NIC’s buffer descriptor rings.

- “**Streaming**”
  - Typically mapped for one data transfer, unmapped afterwards. (*)
  - Hardware can optimize for sequential accesses.
  - “asynchronous”, “outside the coherency domain”.
  - Typical use in a network driver: for data buffers transmitted/received by the device.

(*) high-performance drivers reuse mappings with page pool, https://docs.kernel.org/networking/page_pool.html.

Device's DMA addressing range

- Some devices can address 64 bits, some only 32. Very old devices even less (24).
- The driver must inform the kernel about its device's DMA addressing ability.

```
int dma_set_mask_and_coherent(struct device *dev, u64 mask);
```

Example:

```
r = dma_set_mask_and_coherent(&pdev->dev, DMA_BIT_MASK(64));
```

- This sets the same range for streaming and coherent mappings. There are functions available for setting them distinctly if needed.

Using consistent DMA

- Allocate DMA coherent memory with

```
void *dma_alloc_coherent(struct device *dev, size_t size,
                         dma_addr_t *dma_handle, gfp_t gfp);
```

  The return value is a pointer to the allocated buffer, for use from the CPU side.
  dma_handle is an output parameter: the bus address, for use from the device side.

- Free it with **`dma_free_coherent`**`(dev, size, cpu_addr, dma_handle)`.

Example: Using consistent DMA

- This is called from *cp_open()*, on `ip link set … up`:

```
static int cp_alloc_rings (struct cp_private *cp)
{
    struct device *d = &cp->pdev->dev;
    void *mem;
    int rc;

    mem = dma_alloc_coherent(d, CP_RING_BYTES,
                &cp->ring_dma, GFP_KERNEL);
    if (!mem)
        return -ENOMEM;

    cp->rx_ring = mem;
    cp->tx_ring = &cp->rx_ring[CP_RX_RING_SIZE];
    rc = cp_init_rings(cp);
    if (rc < 0)
        dma_free_coherent(d, CP_RING_BYTES,
                cp->rx_ring, cp->ring_dma);
    return rc;
}
```

- Later, in function *cp_start_hw()*, the driver tells the NIC the addresses of the ring buffers:

```
dma_addr_t ring_dma;
/* … */
ring_dma = cp->ring_dma;
cpw32_f(RxRingAddr, ring_dma & 0xffffffff);
cpw32_f(RxRingAddr + 4, (ring_dma>>16)>>16);

ring_dma += sizeof(struct cp_desc) *
                            CP_RX_RING_SIZE;
cpw32_f(TxRingAddr, ring_dma & 0xffffffff);
cpw32_f(TxRingAddr + 4, (ring_dma>>16)>>16);
```

- Now the device and the driver can see the same descriptor rings.

Streaming DMA direction

- For streaming mappings, the driver must specify the direction of the data flow:

```
enum dma_data_direction {
    DMA_BIDIRECTIONAL = 0,  // will work universally, but may be slower
    DMA_TO_DEVICE = 1,      // from main RAM to device
    DMA_FROM_DEVICE = 2,    // from device to main RAM
    DMA_NONE = 3,           // only for debugging or driver's internal use
};
```

- In a network driver:
  transmitting packets => `DMA_TO_DEVICE`
  receiving packets => `DMA_FROM_DEVICE`

Using streaming DMA

- Memory allocation
  - There is no special API for it.
  - Memory from the page allocator (*__get_free_pages()*) or from the generic memory allocators (*kmalloc()*, *kmem_cache_alloc()*) can be used for DMA.
  - Block I/O and networking subsystems make sure that the buffers they pass to drivers can be used for DMA.
- Create the mapping with:

```
dma_addr_t dma_map_single(struct device *dev, void *ptr,
                          size_t size, enum dma_data_direction dir);
```

  All 4 arguments are input.

  The return value (let's call it *dma_handle*) must be checked with:

```
if (dma_mapping_error(dev, dma_handle)) { /* … error …*/ }
```

- Free it with ***dma_unmap_single()***.
- Other DMA APIs: *dma_{un,}map_sg()*, *dma_sync_*()*

Example: Using streaming DMA

- In *cp_start_xmit()*, starting packet Tx:

```
struct cp_desc *txd = &cp->tx_ring[entry];
len = skb->len;
mapping = dma_map_single(&cp->pdev->dev,
            skb->data, len, DMA_TO_DEVICE);
if (dma_mapping_error(&cp->pdev->dev,
                        mapping))
    goto out_dma_error;
/* … */
txd->addr = cpu_to_le64(mapping);
wmb();
opts1 = len | /* … */;
txd->opts1 = cpu_to_le32(opts1);
wmb();
cp->tx_opts[entry] = opts1;
/* … */
cpw8(TxPoll, NormalTxPoll);
```

- When Tx is done, in *cp_tx()* called from *cp_interrupt()*:

```
struct cp_desc *txd = cp->tx_ring + tx_tail;

/* … */
dma_unmap_single(&cp->pdev->dev,
    le64_to_cpu(txd->addr),
    cp->tx_opts[tx_tail] & 0xffff, // len
    DMA_TO_DEVICE);
```

Network drivers

Topics specific to network drivers

- We saw parts of *8139cp* illustrating what a PCI device driver does.
- Let's look at how the driver does network specific tasks.
  - net device allocation and registration
  - setting the MAC address, MTU, Rx mode, offload features
  - Tx and related features: scatter-gather, Tx checksum, TCP segmentation offload (TSO), VLAN tag insertion
  - Rx, NAPI, and related features: Rx checksum, VLAN tag stripping
  - ethtool settings
- Newer hardware has many more capabilities and needs more complicated drivers.
  - SRIOV, devlink, switchdev, PTP, RDMA, tc offload, ...

Net device allocation

- The PCI driver's *.probe* method (*cp_init_one()*) allocates a *struct net_device* instance, together with a private struct:

```
struct net_device *dev;
struct cp_private *cp;
dev = alloc_etherdev(sizeof(struct cp_private));
if (!dev)
   return -ENOMEM;
SET_NETDEV_DEV(dev, &pdev->dev);
cp = netdev_priv(dev);
cp->pdev = pdev;
cp->dev = dev;
/* ... initialize more cp fields ... */
```

Net device registration

- The *.probe* method then initializes fields in *struct net_device*.
  Pointers to the driver's *net_device_ops* and *ethtool_ops* method tables:
  ```
  dev->netdev_ops = &cp_netdev_ops;
  dev->ethtool_ops = &cp_ethtool_ops;
  ```
  More fields: *features* (features turned on), *hw_features* (toggleable features), *watchdog_timeo* (Tx timeout length), *min_mtu*, *max_mtu*.
- It registers the net device:
  ```
  rc = register_netdev(dev);
  ```
  The net device now appears in `ip link` and may be opened from userspace.
- It saves the net device pointer as driver data in *struct pci_dev*, so it can retrieve it in other *pci_driver* methods (*.remove* and power management methods):
  ```
  pci_set_drvdata(pdev, dev);
  ```
- In *.remove* (*cp_remove_one()*): ***unregister_netdev()***, ***free_netdev()***.

net_device_ops

```
static const struct net_device_ops cp_netdev_ops = {
    .ndo_open             = cp_open,             // ip link set … up
    .ndo_stop             = cp_close,            // ip link set … down
    .ndo_validate_addr    = eth_validate_addr,   // Is the configured MAC valid?
    .ndo_set_mac_address  = cp_set_mac_address, // Configure MAC address
    .ndo_set_rx_mode      = cp_set_rx_mode,      // Promisc mode, unicast, multicast filters
    .ndo_get_stats        = cp_get_stats,        // Get statistics (ip -s link)
    .ndo_eth_ioctl        = cp_ioctl,            // Only for MII. Don't add custom ioctls.
    .ndo_start_xmit       = cp_start_xmit,       // Send a packet
    .ndo_tx_timeout       = cp_tx_timeout,       // netdev watchdog: tx queue timed out
    .ndo_set_features     = cp_set_features,     // ethtool -K … {on,off}            [1]
    .ndo_change_mtu       = cp_change_mtu,       // Configure MTU
    .ndo_features_check   = cp_features_check,   // each Tx skb, may refuse TSO,…
    .ndo_poll_controller  = cp_poll_controller,  // for netconsole                   [1]
}
```

[1] buggy in 8139cp in v6.12

Red Hat

Net device features, userspace view

```
$ ethtool -k enp1s0 | grep -vF 'off [fixed]'
Features for enp1s0:
rx-checksumming: on
tx-checksumming: on
    tx-checksum-ipv4: on
scatter-gather: on
    tx-scatter-gather: on
tcp-segmentation-offload: on
    tx-tcp-segmentation: on
    tx-tcp-mangleid-segmentation: off
generic-segmentation-offload: on
generic-receive-offload: on
rx-vlan-offload: on
tx-vlan-offload: on
highdma: on [fixed]
tx-nocache-copy: off
rx-gro-list: off
rx-udp-gro-forwarding: off
```

Net device features, in kernel

- *struct net_device* has several fields of type *netdev_features_t*. Mainly:
  - *features* – currently enabled features
  - *hw_features* – toggleable features
  - *wanted_features* – features requested to be enabled, but not necessarily enabled right now (you may want TSO on, but tx-checksumming is off).
  - *vlan_features*, *hw_enc_features*, … – features supported for VLAN-tagged, hw-encapsulated packets, …
  - Feature bits: *NETIF_F_**
- *ndo_** callbacks related to features:
  - ```
    netdev_features_t (*ndo_fix_features)(struct net_device *dev,
                                          netdev_features_t features);
    ```
    - The driver may return a subset of *features*, thus rejecting some, based on its hw constraints.
  - ```
    int (*ndo_set_features)(struct net_device *dev, netdev_features_t features);
    ```
    - Commit the new feature configuration to the hardware.
  - ```
    netdev_features_t (*ndo_features_check)(struct sk_buff *skb,
                            struct net_device *dev, netdev_features_t features);
    ```
    - Called before packet Tx. Driver may reject features for the *skb*.
      E.g. *cp_features_check()* rejects *NETIF_F_TSO* if the requested segment size is too big.
- https://docs.kernel.org/networking/netdev-features.html
    ```

Tx feature: scatter-gather

- *NETIF_F_SG*
- "tx-scatter-gather" in *ethtool -k*
- A driver with this feature must be able to transmit non-linear SKBs.
  - SKBs that have fragments.
  - *skb_shinfo(skb)->**frags**[]* - array of *skb_frag_t* structures.
  - *skb_shinfo(skb)->**nr_frags*** > 0
  - *skb->data_len* > 0
  - *skb_headlen(skb)*, the length of the skb's linear part, is *skb->len* - *skb->data_len*.
  - Each frag has:
    - A *struct page**, where the frag's data is. *skb_frag_page(frag)*.
    - An offset within the page where the data begins. *skb_frag_off(frag)*. *skb_frag_address(frag)* gives a pointer to the data.
    - A size. *skb_frag_size(frag)*.
- *8139cp* driver, in *cp_start_xmit()*:
  - Maps the skb's linear part for DMA first, then each frag.
  - Writes a Tx descriptor for each frag and finally for the linear part.

Tx feature: Tx checksum offload

- *NETIF_F_HW_CSUM, NETIF_F_IP_CSUM, NETIF_F_IPV6_CSUM*
- "tx-checksumming" in *ethtool -k*
- The network stack requests checksum offload for a packet by passing it to the driver with *skb->ip_summed == CHECKSUM_PARTIAL*
- https://docs.kernel.org/networking/skbuff.html#checksum-information

- *8139cp* advertises *NETIF_F_IP_CSUM*. In *cp_start_xmit()* it handles Tx checksum offload by setting appropriate bits in the descriptor:

```
if (skb->ip_summed == CHECKSUM_PARTIAL) {
    const struct iphdr *ip = ip_hdr(skb);
    if (ip->protocol == IPPROTO_TCP)
        opts1 |= IPCS | TCPCS;
    else if (ip->protocol == IPPROTO_UDP)
        opts1 |= IPCS | UDPCS;
    else { /* … error … */ }
 }
```

Tx feature: VLAN tag insertion

- *NETIF_F_HW_VLAN_CTAG_TX*
- "tx-vlan-offload" in *ethtool -k*
- A *skb* has two VLAN-related members:
  - *vlan_proto* - a.k.a. Tag Protocol Identifier (TPID). Usually 0x8100 (big endian).
  - *vlan_tci* - Tag Control Information (TCI). The VLAN ID (0–4095) is a part of it.
- https://en.wikipedia.org/wiki/IEEE_802.1Q
- *skb_vlan_tag_get(skb)* returns the *vlan_tci*.
- The network stack requests VLAN tag insertion by passing to the driver a *skb* for which *skb_vlan_tag_present(skb)* is true.

- *8139cp*, in *cp_start_xmit()*, puts the appropriate information in the Tx descriptor:

```
opts2 = cpu_to_le32(cp_tx_vlan_tag(skb));
```

- `static inline u32 cp_tx_vlan_tag(struct sk_buff *skb)`

```
{
    return skb_vlan_tag_present(skb) ?
        TxVlanTag | swab16(skb_vlan_tag_get(skb)) : 0x00;
}
```

Tx feature: TCP Segmentation Offload (TSO)

- *NETIF_F_TSO*
- “tcp-segmentation-offload” in *ethtool -k*
- The stack requests TSO by passing a *skb* with non-zero GSO fields to the driver.
- *skb_shinfo(skb)->gso_type* - bits specify the type of segmentation to be done (*SKB_GSO_TCPV4*, …)
- *skb_shinfo(skb)->gso_size* - how much data to put in each segment
- https://docs.kernel.org/networking/segmentation-offloads.html

- *8139cp*, in *cp_start_xmit()*, puts the appropriate information in the Tx descriptor:

```
mss = skb_shinfo(skb)->gso_size;
/* … */
if (mss)
   opts1 |= LargeSend | (mss << MSSShift);
```


Red Hat

Tx completion

- When the NIC finishes sending a packet, the driver
  - unmaps the DMA mapping,
  - consumes the *skb*,
  - wakes up the Tx queue if sufficient descriptors became available.
    - whereas stopping of the queue is done in *.ndo_start_xmit* when the number of available descriptors gets low.
- Some drivers do Tx completion in their interrupt handlers.
  - Like *8139cp*: *cp_interrupt()* -> *cp_tx()*.
- Some drivers do it in NAPI poll.

Red Hat

Rx with NAPI

- DMA buffers for Rx have to be set up beforehand.
  - *8139cp*: *cp_refill_rx()* allocates SKBs, DMA maps them, puts their addresses in the Rx ring.
- When the NIC receives packets, it writes them to the buffers, writes information to the Rx descriptors and triggers an interrupt.
- The driver's interrupt handler masks the Rx interrupt and schedules NAPI poll.
  - *8139cp*: cp_interrupt() writes to the *IntrMask* register, uses *napi_schedule_prep()* and *__napi_schedule()*.
- NAPI poll runs in bottom half context,
  - processes completed items in the Rx ring,
  - allocates replacement SKBs and DMA maps them,
  - unmaps the completed buffers,
  - fills SKB metadata (*skb->{protocol, pkt_type, len, ...}*, uses *eth_type_trans()*),
  - passes the received SKBs to the stack,
  - calls *napi_complete_done()* and unmasks the NIC's Rx interrupts when no more work.
  - *8139cp*'s NAPI poll routine is *cp_rx_poll()*.
- https://docs.kernel.org/networking/napi.html

Rx feature: Rx checksum offload

- The NIC may calculate the Rx checksum and
  - report the result in the Rx descriptor, or
  - indicate in the Rx descriptor that it verified the checksum.

- A driver for the former case would set:

```
skb->csum = /* the NIC-reported checksum value */;
skb->ip_summed = CHECKSUM_COMPLETE;
```

- *8139cp* is the latter case.

  Its Rx descriptor has bits to indicate checksum failures for TCP and UDP packets.

```
if (cp_rx_csum_ok(status))
    skb->ip_summed = CHECKSUM_UNNECESSARY;
else
    skb_checksum_none_assert(skb);    // leaves ip_summed==CHECKSUM_NONE
```

Rx feature: VLAN tag stripping

- The NIC may have the ability to remove (strip) the VLAN tag from the received packets' data and instead report the VLAN tag as metadata in the Rx descriptor.

- *8139cp*: *cp_rx_skb()*:

```
u32 opts2 = le32_to_cpu(desc->opts2);
/* … */
if (opts2 & RxVlanTagged)
   __vlan_hwaccel_put_tag(skb, htons(ETH_P_8021Q),
                          swab16(opts2 & 0xffff));

napi_gro_receive(&cp->napi, skb);
```

ethtool

- Many network device settings can be queried or set using *ethtool*.
- `static const struct ethtool_ops cp_ethtool_ops = {`

```
    .get_drvinfo     = cp_get_drvinfo,             // Get driver info, ethtool -i
    .get_regs_len    = cp_get_regs_len,            // Get size of register dump, ethtool -d
    .get_sset_count  = cp_get_sset_count,          // Get the number of strings (stats names)
    .nway_reset      = cp_nway_reset,              // Restart link auto-negotiation
    .get_link        = ethtool_op_get_link,        // Get link state
    .get_msglevel    = cp_get_msglevel,            // Get configuration of debug messages
    .set_msglevel    = cp_set_msglevel,            // Set configuration of debug messages
    .get_regs        = cp_get_regs,                // Dump registers, ethtool -d
    .get_wol         = cp_get_wol,                 // Get Wake-on-LAN settings
    .set_wol         = cp_set_wol,                 // Set Wake-on-LAN settings
    .get_strings     = cp_get_strings,             // Return a set of strings (stats names)
    .get_ethtool_stats = cp_get_ethtool_stats,     // Get statistics, ethtool -S
    .get_eeprom_len  = cp_get_eeprom_len,          // ethtool --eeprom-dump
    .get_eeprom      = cp_get_eeprom,              // ethtool --eeprom-dump
    .set_eeprom      = cp_set_eeprom,              // ethtool --change-eeprom
    .get_ringparam   = cp_get_ringparam,           // Get ring parameters, ethtool -g
    .get_link_ksettings = cp_get_link_ksettings,   // Get ethtool speed, duplex, autoneg, …
    .set_link_ksettings = cp_set_link_ksettings,   // Set ethtool speed, duplex, autoneg, …
};
```

Learn more

- Linux Device Drivers book available in PDFs
  - https://lwn.net/Kernel/LDD3/
  - Old, but still useful.

Thank you

Red Hat is the world's leading provider of enterprise open source software solutions. Award-winning support, training, and consulting services make Red Hat a trusted adviser to the Fortune 500.

linkedin.com/company/red-hat

youtube.com/user/RedHatVideos

facebook.com/redhatinc

twitter.com/RedHat

Red Hat