

Lecture 14.2

The Device Driver Model
buses, devices, drivers, sysfs

Md Mahbub E Noor

Assistant Professor, Department of CSE

University of Barishal

```
● ● ● readlink -f /sys/class/block/vda (real)
/sys/devices/pci0000:00/
  0000:00:04.0/virtio0/
  block/vda
# a disk, on virtio, on PCI
```

What you will learn today



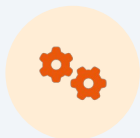
Explain device files, major and minor numbers, and devtmpfs.



Find a device's bus, parent and driver in /sys, and name the kernel structs behind them.



Explain how a bus matches a driver to a device and calls probe; bind and unbind by hand.



Write a platform driver with a sysfs file, and follow its uevents.

Today's plan (60 min)

0-5

Recap of 14.1

5-12

Device files

12-25

The device model, /sys

25-35

Match and probe

35-48

A platform driver

48-53

Uevents

53-60

Lab, checkpoint

Lab 14.2: sp_drv.c, a driver with a sysfs file.

Quick recap of Lecture 14.1

Q1

Who fills the U _printk hole in hello.ko?

The kernel's module loader: it finds _printk in its symbol table and patches the call (R_X86_64_PLT32).

Q2

hello_init returned -EINVAL. Is hello_exit called?

No: a failed init means the module was never loaded. Init must undo its own work.

Q3

Why did rmmod counter fail while user.ko was loaded?

user uses counter's exported symbol: refcount 1 ("Used by user").

PART 1

Devices, seen from user space

A file in /dev, two numbers, and a driver behind them.

A device file is a name for (type, major, minor)

```
ls -l /dev/null /dev/ttyS0 /dev/vda (QEMU guest, real)
```

```
crw-rw-rw- 1 0 0 1, 3 /dev/null
crw----- 1 0 0 4, 64 /dev/ttyS0
brw----- 1 0 0 253, 0 /dev/vda
```

```
/proc/devices (real)
```

```
Character devices:
 1 mem
 4 ttyS
 5 /dev/tty
10 misc
136 pts
Block devices:
 8 sd
253 virtblk
```

```
kdev_t.h (real)
```

```
7 #define MINORBITS 20
10 #define MAJOR(dev) \
    ((dev) >> MINORBITS)
11 #define MINOR(dev) \
    ((dev) & MINORMASK)
12 #define MKDEV(ma,mi) \
    (((ma) << MINORBITS) | (mi))
```

c / b: character device (a stream of bytes: terminal, /dev/null) or block device (a disk of blocks, through the page cache)

major: which driver: 1 = mem, 4 = ttyS, 253 = virtblk

minor: which one of its devices: /dev/null is mem's number 3; ttyS0 is ttyS's 64

dev_t: both numbers packed in one 32-bit integer: 12 bits major, 20 bits minor

Book: LDD3 Ch. 3 (major and minor numbers); Love, LKD Ch. 13–14

PART 2

The device model and /sys

One tree of every device, every driver, and how they connect.

Four objects describe all hardware: bus, device, driver, class

object	is	example in our guest	in /sys
bus_type	a kind of connection, with a match rule	pci, virtio, platform, usb	/sys/bus/*
device	one thing that exists	0000:00:04.0, virtio0, vda, sysprog.0	/sys/devices/...
device_driver	code that can run some devices	virtio-pci, virtio_blk, sysprog	/sys/bus/*/drivers/*
class	devices by what they do, not how attached	block, net, tty, input	/sys/class/*

Analogy: a job fair. Each hall (bus) has a list of companies (drivers) and a list of applicants (devices). The hall's own rule decides who fits whom; when a pair fits, the interview (probe) happens. The phone book (class) lists people by job, not by hall.

SAME AS C

Structs and lists

Each object is a C struct (device.h, device/bus.h, device/driver.h) and each bus keeps two linked lists: its devices and its drivers. sysfs shows those lists as directories.

Book: LDD3 Ch. 14 (the Linux device model); Love, LKD Ch. 17

/sys is that model, as directories and files

```
ls /sys/bus (QEMU guest, real)
```

```
acpi      auxiliary  clockevents  clocksource
container  cpu        dax          edac
event_source  gpio      i2c          isa
machinecheck  mdio_bus  memory       mmc
pci       platform  pnp          scsi
serial    serio     usb          virtio ...
```

```
ls /sys/bus/pci/drivers (real, cut)
```

```
8250_mid  ata_generic  ata_piix  ehci-pci
ohci-pci  pcieport    serial    shpchp  uhci_hcd
virtio-pci xen-platform-pci ...
# drivers BUILT INTO this kernel: no modules loaded
```

```
the disk, three ways (real)
```

```
$ ls -l /sys/class/block/ | grep vda
vda -> ../../devices/pci0000:00/0000:00:04.0/
      virtio0/block/vda
$ readlink -f /sys/class/block/vda/device/driver
/sys/bus/virtio/drivers/virtio_blk
$ readlink -f /sys/bus/pci/devices/0000:00:04.0/driver
/sys/bus/pci/drivers/virtio-pci
```

1 /sys/devices: the real tree, by parent (who is attached to whom)

2 /sys/bus/X/{devices,drivers}: symlinks into it, per bus

3 /sys/class/X: symlinks again, by function

Book: LDD3 Ch. 14; Love, LKD Ch. 17 (sysfs)

Follow the disk: two buses, two drivers, one block device

PCI device 0000:00:04.0

vendor 0x1af4, device 0x1001, class 0x010000 (storage)

driver: virtio-pci (bus pci)

virtio device virtio0

created by virtio-pci; device id 2 = block

driver: virtio_blk (bus virtio)

block device vda

created by virtio_blk; class block

dev 253:0 → /dev/vda

the IDs the buses match on (real)

pci modalias: pci:v00001AF4d00001001sv00001AF4sd00000002bc01sc00i00

virtio modalias: virtio:d00000002v00001AF4

vda uevent: MAJOR=253 MINOR=0 DEVNAME=vda DEVTYPE=disk

drivers/block/virtio_blk.c (real)

```
1627 static const struct
      virtio_device_id id_table[] = {
1628     { VIRTIO_ID_BLOCK, VIRTIO_DEV_ANY_ID },
1629     { 0 },
1630 };
1648 static struct virtio_driver virtio_blk = {
1655     .id_table = id_table,
1656     .probe    = virtblk_probe,
1657     .remove   = virtblk_remove,
```

VIRTIO_ID_BLOCK = 2 (virtio_ids.h line 33): "d00000002" in the modalias. That is the match.

The C structs behind it (Linux 6.8)

```
● ● ● include/linux/device/bus.h

77 struct bus_type {
78     const char *name;
84     int (*match)(struct device *,
                   struct device_driver *);
85     int (*uevent)(...);
86     int (*probe)(struct device *);
88     void (*remove)(struct device *);
```

```
● ● ● include/linux/device/driver.h

96 struct device_driver {
97     const char *name;
98     const struct bus_type *bus;
106    const struct of_device_id *...;
109    int (*probe)(struct device *);
111    int (*remove)(struct device *);
116    const struct attribute_group
        **dev_groups;
```

```
● ● ● include/linux/device.h

703 struct device {
704     struct kobject kobj;
705     struct device *parent;
712     const struct bus_type *bus;
713     struct device_driver *driver;
717     void *driver_data;
776     dev_t devt;
782     const struct class *class;
```

SAME AS C

Function pointers, once more

bus_type and device_driver are tables of callbacks, like file_operations (12.1) and irq handlers (13).

DIFFERENT FROM C

"Inheritance" by embedding

struct platform_device contains a struct device; container_of gets back from the inner to the outer struct.

Book: LDD3 Ch. 14; Love, LKD Ch. 17

PART 3

Match and probe

Who decides that this driver runs that device?

The bus decides: platform_match

```
● ● ● drivers/base/platform.c (real lines)
1335 static int platform_match(struct device *dev,
                             struct device_driver *drv)
...
1341     if (pdev->driver_override)
1342         return !strcmp(pdev->driver_override, drv->name);
1345     if (of_driver_match_device(dev, drv)) /* device tree */
1349     if (acpi_driver_match_device(dev, drv)) /* ACPI */
1353     if (pdrv->id_table)
1354         return platform_match_id(pdrv->id_table, pdev) != NULL;
1357     return (strcmp(pdev->name, drv->name) == 0);
```

bus	matches by
platform	name (or device tree / ACPI / id table)
pci	vendor, device, class ids in the driver's id_table
usb	vendor, product, interface class (Lecture 15.3)
virtio	device id (2 = block) and vendor

The platform bus is for devices that cannot be discovered by asking the hardware (no PCI ids): SoC blocks on phones and boards, the PC's serial port (serial8250) and keyboard controller (i8042). Someone must register them: firmware tables, or code like our sp_dev.ko.

When is match called? Whenever a device is added (try every driver of the bus) and whenever a driver is added (try every device without a driver). So the load order does not matter: our run proves it both ways.

Book: LDD3 Ch. 14 (buses, devices, drivers)

A match: really_probe calls the driver's probe

```
● ● ● drivers/base/dd.c and platform.c (real lines)

dd.c 603 static int really_probe(struct device *dev,
                                struct device_driver *drv)
    645     ret = driver_sysfs_add(dev); /* driver symlinks */
    572 static int call_driver_probe(...)
    577     ret = dev->bus->probe(dev);
plat 1379 static int platform_probe(struct device *_dev)
    1404     ret = drv->probe(dev); /* our probe */
dd.c 677     ret = device_add_groups(dev, drv->dev_groups);
    421     kobject_uevent(&dev->kobj, KOBJ_BIND);
```

probe's job: check the device really works, allocate per-device state, map registers, request IRQs (Lecture 13), create user interfaces

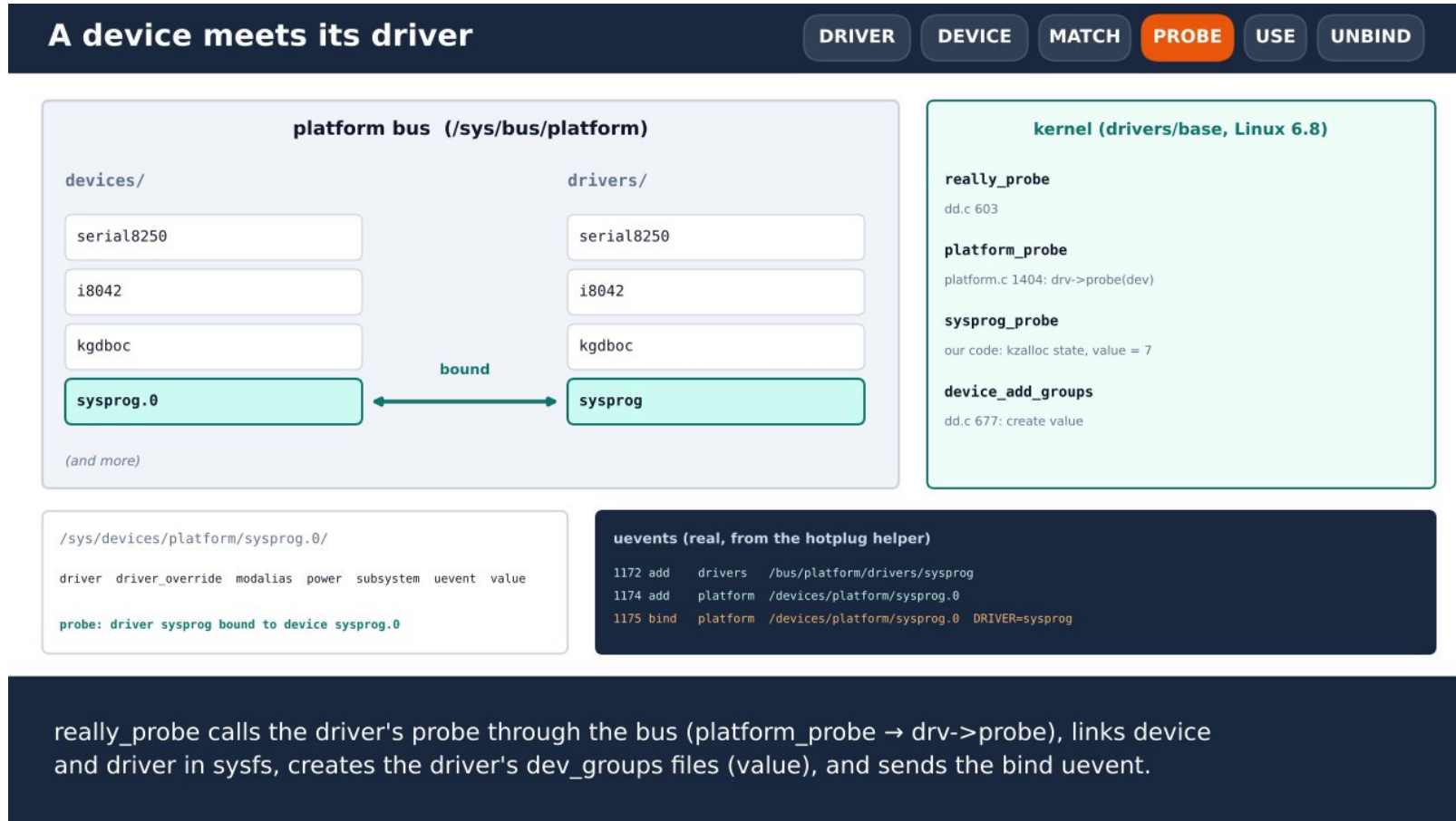
returns 0: bound: dev->driver is set, the driver symlink appears, sysfs files from dev_groups are created, a bind uevent goes out

returns < 0: not bound; the bus may try another driver. -EPROBE_DEFER: try again later (a resource is not ready yet)

Probe is where a driver meets ONE device. A driver module is loaded once; probe runs once per matching device (our sp_dev2 with count=2: probe ran for sysprog.0 and sysprog.1, each with its own state). remove is its mirror: undo what probe did for that device.

Book: LDD3 Ch. 14 (device and driver binding)

Watch a device meet its driver



Click the video to play or pause. About 1.2 minutes, no sound.

Watch for

1. The driver **waits** for a device.
2. **strcmp** of the two names.
3. probe: state, sysfs **value**.
4. **uevents**: add, add, bind.
5. **unbind** runs remove.

PART 4

A platform driver, line by line

`sp_dev.ko` registers a device; `sp_drv.ko` drives it (Lab 14.2).

Two small modules: a device and its driver

● ● ● sp_dev.c (core lines)

```
static struct platform_device *pdev;
/* init: */
pdev = platform_device_register_simple(
    "sysprog", 0, NULL, 0); /* sysprog.0 */
if (IS_ERR(pdev))
    return PTR_ERR(pdev);
/* exit: */
platform_device_unregister(pdev);
```

IS_ERR / PTR_ERR a pointer that carries an errno: kernel style instead of NULL + errno

devm_kzalloc zeroed memory owned by the device: freed automatically after remove

module_platform_driver writes init/exit that only register/unregister the driver

● ● ● sp_drv.c (core lines)

```
static int sysprog_probe(struct platform_device *pdev)
{
    st = devm_kzalloc(&pdev->dev, sizeof(*st), GFP_KERNEL);
    if (!st) return -ENOMEM;
    mutex_init(&st->lock);
    st->value = 7;
    platform_set_drvdata(pdev, st);
    return 0;
}

static struct platform_driver sysprog_driver = {
    .probe      = sysprog_probe,
    .remove_new = sysprog_remove, /* 6.8 */
    .driver = { .name      = "sysprog",
                .dev_groups = sysprog_groups },
};

module_platform_driver(sysprog_driver);
```

Load them in either order: the bus binds them (real)

● ● ● driver first (QEMU guest)

```
$ insmod sp_drv.ko; ls /sys/bus/platform/drivers/sysprog
bind module uevent unbind
$ insmod sp_dev.ko; dmesg | tail -2
sysprog sysprog.0: probe: driver sysprog bound
                        to device sysprog.0
sp_dev: device sysprog.0 registered
$ readlink /sys/devices/platform/sysprog.0/driver
../../../../bus/platform/drivers/sysprog
```

● ● ● device first (QEMU guest)

```
$ insmod sp_dev.ko
$ ls /sys/devices/platform/sysprog.0
driver_override modalias power
subsystem uevent # no driver
$ insmod sp_drv.ko; dmesg | tail -1
sysprog sysprog.0: probe: driver sysprog bound
                        to device sysprog.0
```

● ● ● after binding (real)

```
$ ls /sys/devices/platform/sysprog.0
driver driver_override modalias power subsystem uevent value # driver and value are new
$ cat /sys/devices/platform/sysprog.0/uevent
DRIVER=sysprog MODALIAS=platform:sysprog
```

A sysfs file: show and store functions

sp_drv.c (core lines)

```
static ssize_t value_show(struct device *dev,
                          struct device_attribute *attr, char *buf)
{
    struct sysprog_state *st = dev_get_drvdata(dev);
    ... v = st->value; ... /* under st->lock */
    return sysfs_emit(buf, "%d\n", v);
}

static ssize_t value_store(struct device *dev, ...,
                           const char *buf, size_t count)
{
    int v, err = kstrtoint(buf, 10, &v);
    if (err) return err; /* "abc": -EINVAL */
    ... st->value = v; st->writes++; ...
    return count;
}

static DEVICE_ATTR_RW(value); /* dev_attr_value */
```

QEMU guest (real)

```
$ cd /sys/devices/platform/sysprog.0
$ cat value
7
$ echo 42 > value; cat value
42
$ echo abc > value; echo $?
sh: write error: Invalid argument
1
```

SAME AS C

Getter and setter

In C you would write `int get_value(struct dev *)` and `void set_value(struct dev *, int)`. sysfs calls yours when someone reads or writes the file. Rule: one value per file, as text.

Unbind and bind by hand; one state per device

QEMU guest (real)

```
$ echo sysprog.0 > /sys/bus/platform/drivers/sysprog/unbind
sysprog sysprog.0: remove: last value 42 after 1 writes
$ ls /sys/devices/platform/sysprog.0
driver_override power uevent modalias subsystem
$ echo sysprog.0 > /sys/bus/platform/drivers/sysprog/bind
sysprog sysprog.0: probe: driver sysprog bound ...
$ cat /sys/devices/platform/sysprog.0/value
7
# a NEW state: probe ran again
```

Unbind: runs remove, deletes the driver's sysfs files, frees the devm memory. The device stays.

Bind: runs probe again: a fresh state (value 7).

two devices: insmod sp_dev2.ko count=2 (real)

```
sysprog sysprog.0: probe: driver sysprog bound ...
sysprog sysprog.1: probe: driver sysprog bound ...
$ echo 5 > .../sysprog.1/value; cat .../sysprog.0/value .../sysprog.1/value
7
5
```

Per device: state lives in driver_data, not in a global variable: two devices, two values.

C link: a C program often keeps state in globals; a driver must not: it may serve many devices at once.

PART 5

Uevents: the kernel tells user space

Every add, bind, unbind and remove, as a message.

Uevents: what udev hears (real, from our guest)

every uevent of insmod sp_drv, insmod sp_dev, unbind, bind, rmmmod ...

```
SEQNUM=1172 ACTION=add SUBSYSTEM=drivers DEVPATH=/bus/platform/drivers/sysprog
SEQNUM=1173 ACTION=add SUBSYSTEM=module DEVPATH=/module/sp_drv
SEQNUM=1174 ACTION=add SUBSYSTEM=platform DEVPATH=/devices/platform/sysprog.0
MODALIAS=platform:sysprog
SEQNUM=1175 ACTION=bind SUBSYSTEM=platform DEVPATH=/devices/platform/sysprog.0
DRIVER=sysprog
SEQNUM=1176 ACTION=add SUBSYSTEM=module DEVPATH=/module/sp_dev
SEQNUM=1177 ACTION=unbind SUBSYSTEM=platform DEVPATH=/devices/platform/sysprog.0
SEQNUM=1178 ACTION=bind SUBSYSTEM=platform DEVPATH=/devices/platform/sysprog.0
... 1179 unbind, 1180-1183 remove (driver, module, device, module)
```

1

the kernel builds KEY=value lines
(lib/kobject_uevent.c)

2

sends them on a netlink socket (udev
listens)

3

or runs a helper program: our script just
logged them

On a real system, udev reacts: a device with MODALIAS appears → `modprobe platform:sysprog` loads the matching driver (modules list their aliases in `modules.alias`); a device with MAJOR/MINOR appears → a `/dev` node with the right owner and permissions, and nice names like `/dev/disk/by-uuid/...`

Book: LDD3 Ch. 14 (hotplug); Love, LKD Ch. 17

LAB 14.2

sp_drv.c: a driver with a sysfs file

probe, remove, show and store; then bind, unbind and break it on purpose.

Lab 14.2 tasks, step by step

1 `cd lab14_2; make; ../guest/run_guest.sh` . then `insmod sp_dev.ko` and `sp_drv_starter.ko`: **no probe** (why?)

2 **TODO 4** first: the driver's `.name` must match the device's name

3 **TODO 3**: probe: `devm_kzalloc` the state, `mutex_init`, `value = 7`, `set_drvdata`

4 **TODO 1, 2**: `value_show` with `sysfs_emit`; `value_store` with `kstrtoint`

5 Test: `cat`, `echo 42`, `echo abc`, `unbind`, `bind`, `rmmod` in both orders

The starter loads but never binds (its driver is called `sysprog-TODO`). Finding out why, with `ls /sys/bus/platform/drivers` and `dmesg`, is part of the lab.

● ● ● the starter, before **TODO 4** (real)

```
$ insmod sp_dev.ko; insmod sp_drv_starter.ko
$ ls /sys/bus/platform/drivers
ac  acpi-fan  ...  syscon
sysprog-TODO  tpm_tis  ...
$ ls /sys/devices/platform/sysprog.0
driver_override  power  uevent
modalias  subsystem          # no driver!
# after the lab:
$ cat /sys/devices/platform/sysprog.0/value
7
$ echo 42 > .../value; cat .../value
42
```

Checkpoint: test yourself

- Q1** `ls -l /dev/vda` shows "b ... 253, 0". What do b, 253 and 0 mean?
→ block device, major 253 (virtblk), minor 0 (the first disk)
- Q2** Where is the only real copy of a device's directory in `/sys`?
→ `/sys/devices/...`; bus and class entries are symlinks
- Q3** How does the platform bus match a driver to a device by default?
→ `strcmp` of the device name and the driver name
- Q4** A driver module is loaded once. How often does probe run?
→ once per matching device (sysprog.0 and sysprog.1)
- Q5** Which uevent variable lets udev load the right driver module?
→ MODALIAS (e.g. platform:sysprog, virtio:d00000002...)

Key ideas from today



Device files name (type, major, minor); major picks the driver, minor the device.



Bus, device, driver, class: C structs with callbacks, shown as /sys directories and links.



The bus's match pairs drivers and devices; probe sets up one device, remove undoes it.



sysfs files call show/store; uevents tell udev to load modules and make /dev nodes.

Homework 14.2 (due before Lecture 15.1)

1. Another name

Build a copy of the driver named "sysprog2". Load it with `sp_dev`: why no probe? Make it bind anyway with `driver_override` (platform.c lines 1341–1342).

2. Two devices

Write `sp_dev2.c`: register count devices (a module parameter). If one registration fails, undo the earlier ones. Load it with `count=2` and give each device a different value.

3. Your disk

On a lab PC: find `/dev/sda`'s (or `nvme0n1`'s) major:minor, its driver, its bus, and the PCI device above it, using only `/sys` (`readlink -f`).

4. Uevent order

In our log SEQNUM 1188 was written before 1187. Which events were they, and why can the helper's log be out of order?

5. The newline

`echo Dhaka > who` (a charp parameter) stored "Dhaka\n", but `echo 42 > value` worked. Find the reason in `kernel/params.c` and `lib/kstrtox.c`.

Next lecture and reading

NEXT: LECTURE 15.1

Character device drivers

Your own /dev file: register a major and minor, write file_operations (open, read, write, release, ioctl), copy data to and from user space, and use it from a C program.

Driver book

Corbet, Rubini & Kroah-Hartman, **LDD3** (free), Ch. 14 today; Ch. 3 (char drivers) for 15.1.

Kernel side

R. Love, **LKD**, Ch. 17 (devices, sysfs, kobjects).

Source

Linux 6.8: drivers/base/{dd,platform,bus,core}.c, include/linux/device*.h, drivers/block/virtio_blk.c (Linux v6.8 source online: elixir.bootlin.com/linux/v6.8).

Lab files

lab14_files.zip: lab14_2/ (sp_dev.c, sp_drv_starter.c), guest/run_guest.sh.