

Lecture 15.3

USB Device Drivers

a keyboard, a URB, and the end of the course

Md Mahbub E Noor

Assistant Professor, Department of CSE

University of Barishal

```
● ● ● dmesg (QEMU guest, real)
sysprog_kbd 1-1:1.0: listening on
        endpoint 0x81
report 1: 00 00 0b 00 ... key 'h'
report 6: 02 00 05 00 ... 'b' +shift
# keys from QEMU's virtual USB keyboard
```

What you will learn today



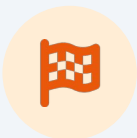
Describe a USB device as descriptors: device, configuration, interface, endpoint.



Explain enumeration, MODALIAS and how usbcore matches a driver's id_table.



Write a USB driver: probe, an interrupt URB, its completion, disconnect.



Place drivers at the end of the path that started with SIC in Lecture 1.

Today's plan (60 min)

0-5

Recap of 15.2

5-15

USB from the kernel

15-25

Matching

25-42

probe, URBs, disconnect

42-50

Lab 15.3

50-60

Course wrap-up

Lab 15.3: sysprog_kbd.c, a USB keyboard driver.

Quick recap of Lecture 15.2

Q1

A write of 4096 bytes at byte 409600 of /dev/sbd0: what does the driver see?

One request: sector 800, 8 sectors, a list of (page, offset, len) segments.

Q2

Why did 8 small writes reach sbd as one request?

They landed in one page of the page cache; sync wrote the page once.

Q3

A real disk driver's queue_rq returns before the data moved. Who ends the request?

The interrupt handler, with blk_mq_complete_request, when the device is done.

PART 1

USB from the kernel's view

A tree of hubs, devices that describe themselves, and a host that does all the asking.

The USB tree in our guest (real)

```
./mysusb (reads /sys/bus/usb/devices; real)

usb1 1d6b:0001 12 Mbit/s ... uhci_hcd / UHCI Host Controller
1-0:1.0 interface class 09/00/00, driver hub
1-1 0627:0001 12 Mbit/s QEMU / QEMU USB Keyboard
1-1:1.0 interface class 03/01/01, driver -
1-2 0409:55aa 12 Mbit/s QEMU / QEMU USB Hub
1-2:1.0 interface class 09/00/00, driver hub
1-2.1 0627:0001 12 Mbit/s QEMU / QEMU USB Mouse
1-2.1:1.0 interface class 03/01/02, driver -
```

name	means
usb1	bus 1: the root hub of the host controller
1-2.1	bus 1, port 2, then port 1 of that hub
1-1:1.0	device 1-1, configuration 1, interface 0

1 the host asks, devices answer: a USB device never sends anything on its own

2 drivers bind to interfaces, not devices: "driver -" = no driver for the keyboard's interface yet

Why does our driver get the keyboard? On a lab PC, usbhid takes it at once. Our guest boots without usbhid (it is a module and the guest has none), so the keyboard is free for us.

Book: LDD3 Ch. 13 (USB device basics)

A device describes itself: the descriptor tree

device 0627:0001, "QEMU USB Keyboard", 1 configuration

configuration 1 power, number of interfaces

interface 0 class 03 HID / subclass 01 boot / protocol 01 keyboard

endpoint 0x81 interrupt, IN, 8 bytes, every 10 ms

SAME AS C

Nested structs

struct usb_device → usb_host_config → usb_interface → usb_host_endpoint: the kernel keeps the tree as C structs, and sysfs shows it as directories.

```
sysfs of the keyboard (real)

$ cd /sys/bus/usb/devices/1-1
$ cat idVendor idProduct speed devnum
0627 0001 12 2
$ cat 1-1:1.0/modalias
usb:v0627p0001d0000dc00dsc00dp00ic03isc01ip01in00
$ cd 1-1:1.0/ep_81; cat type direction ...
type: Interrupt direction: in
wMaxPacketSize: 0008
bInterval: 0a interval: 10ms
```

0x81 = bit 7 set (IN: device → host), endpoint number 1. Endpoint 0 (control) exists on every device: ep_00.

Book: LDD3 Ch. 13 (endpoints, interfaces, configurations)

Four kinds of transfer, four kinds of endpoint

type	for	promise	example
control	setup and status, endpoint 0	delivered, small	reading descriptors, setting an address
interrupt	small, regular data	polled every $bInterval$ ms	keyboard, mouse (today)
bulk	large data, when the bus is free	delivered, no timing	USB stick, printer
isochronous	a steady stream	timing, but no retry	microphone, webcam

"Interrupt" is a misleading name: the device cannot interrupt. The host controller polls the endpoint every 10 ms; only the host controller then interrupts the CPU (Lecture 13) when data came back.

DIFFERENT FROM C

Not like a char device

Nobody calls `read()`: data arrives when the device has some. The driver keeps a request (a URB) waiting at the host controller all the time.

Enumeration: from "something is plugged in" to a device

```
● ● ● dmesg at boot (real)
uhci_hcd 0000:00:04.0: UHCI Host Controller
uhci_hcd 0000:00:04.0: new USB bus registered, assigned bus number 1
uhci_hcd 0000:00:04.0: irq 11, io port 0x0000c040
usb 1-1: new full-speed USB device number 2 using uhci_hcd
usb 1-1: New USB device found, idVendor=0627, idProduct=0001
usb 1-1: New USB device strings: Mfr=1, Product=4, SerialNumber=11
usb 1-1: Product: QEMU USB Keyboard
usb 1-1: Manufacturer: QEMU
```

1 the hub reports a **port change**: hub_event (hub.c 5786)

2 reset the port, give the device **address 2**:
hub_port_connect (5306)

3 read the descriptors on **endpoint 0**: usb_new_device
(2561)

4 add the device and its interfaces to sysfs: **uevents** with
MODALIAS

Lecture 14.2 link: the device appears on the **usb bus** exactly like sysprog.0 appeared on the platform bus: device_add, a uevent, then the bus tries its drivers.

irq 11: the one interrupt line of the host controller. Every keyboard report on the following slides arrives through it.

PART 2

Matching a driver

The `id_table`, the `MODALIAS`, and `usb_probe_interface`.

Which interfaces do we want? The id_table

sysprog_kbd.c (real)

```
static const struct usb_device_id skbd_ids[] = {
    { USB_INTERFACE_INFO(USB_CLASS_HID, 1, 1) },
    /* class 3 HID, subclass 1 boot, protocol 1 keyboard */
    { } /* end of the table */
};
MODULE_DEVICE_TABLE(usb, skbd_ids);
```

include/linux/usb.h (real)

```
1076 #define USB_INTERFACE_INFO(cl, sc, pr) \
    .match_flags = USB_DEVICE_ID_MATCH_INT_INFO, \
    .bInterfaceClass = (cl), \
    .bInterfaceSubClass = (sc), \
    .bInterfaceProtocol = (pr)
```

drivers/usb/core/driver.c (real lines)

```
321 static int usb_probe_interface(struct device *dev)
346     id = usb_match_dynamic_id(intf, driver);
348     id = usb_match_id(intf, driver->id_table);
399     error = driver->probe(intf, id);
677 int usb_match_one_id_intf(...)
692     if ((id->match_flags & USB_DEVICE_ID_MATCH_INT_CLASS) &&
693         (id->bInterfaceClass != intf->desc.bInterfaceClass))
694         return 0; /* and the same for subclass, protocol
*/
```

SAME AS C

A table ended by an empty entry

Like argv's NULL at the end, or a string's 0: the kernel walks skbd_ids until it finds {}. Designated initializers (.bInterfaceClass = ...) leave every other field 0.

MODALIAS: the interface's id card (real)

v0627	p0001	d0000	dc00	dsc00	dp00	ic03	isc01	ip01	in00
vendor (5)	product (5)	bcdDevice (5)	dev class (4)	dsc (5)	dp (4)	if class (4)	if sub (5)	if proto (4)	if no (4)

usb:v0627p0001d0000dc00dsc00dp00ic03isc01ip01in00 (cat /sys/bus/usb/devices/1-1:1.0/modalias)

bind by hand (real)

```
$ echo -n 1-1:1.0 > /sys/bus/usb/drivers/sysprog_kbd/unbind
sysprog_kbd 1-1:1.0: disconnect after 0 reports
$ echo -n 1-1:1.0 > /sys/bus/usb/drivers/sysprog_kbd/bind
sysprog_kbd 1-1:1.0: listening on endpoint 0x81
$ cat /sys/bus/usb/devices/1-2.1:1.0/modalias      # the mouse
usb:v0627p0001d0000dc00dsc00dp00ic03isc01ip02in00
$ echo -n 1-2.1:1.0 > .../sysprog_kbd/bind
sh: write error: No such device
```

1

MODULE_DEVICE_TABLE writes our table into the .ko; depmod turns it into **modules.alias**: `usb:v*p*d*dc*dsc*dp*ic03isc01ip01in*`

2

the uevent carries the MODALIAS; udev runs **modprobe** with it: the right module loads by itself

3

the mouse says **ip02**: our table does not match, bind fails with **ENODEV**

PART 3

probe, URBs, disconnect

Ask the host controller to poll the keyboard, and react when it answers.

struct usb_driver and skbd_probe

● ● ● sysprog_kbd.c (real)

```
static struct usb_driver skbd_driver = {
    .name      = "sysprog_kbd",
    .id_table   = skbd_ids,
    .probe      = skbd_probe,
    .disconnect = skbd_disconnect,
};
module_usb_driver(skbd_driver); /* init/exit for us */
```

● ● ● include/linux/usb.h (real)

```
1212 struct usb_driver {
1213     const char *name;
1215     int (*probe) (struct usb_interface *intf,
                   const struct usb_device_id *id);
1218     void (*disconnect) (struct usb_interface *intf);
1230     const struct usb_device_id *id_table;
```

● ● ● skbd_probe (real, core lines)

```
for (i = 0; i < alt->desc.bNumEndpoints; i++) {
    struct usb_endpoint_descriptor *e = &alt->endpoint[i].desc;
    if (!ep && usb_endpoint_is_int_in(e)) ep = e;
}
if (!ep) return -ENODEV;
k->buf = usb_alloc_coherent(udev, 8, GFP_KERNEL, &k->dma);
k->urb = usb_alloc_urb(0, GFP_KERNEL);
... usb_fill_int_urb(...) (next slide)
usb_set_intfdata(intf, k);
err = usb_submit_urb(k->urb, GFP_KERNEL);
```

● ● ● dmesg after insmod (real)

```
sysprog_kbd 1-1:1.0: probe: 0627:0001 "QEMU" "QEMU USB
Keyboard", interface 0 class 3/1/1, 1 endpoint(s)
sysprog_kbd 1-1:1.0: endpoint 0x81: interrupt IN,
max packet 8, interval 10
sysprog_kbd 1-1:1.0: listening on endpoint 0x81
usbcore: registered new interface driver sysprog_kbd
```

The URB: a request handed to the host controller

```
include/linux/usb.h: struct urb (real, part)

1593 struct urb {
1606     struct usb_device *dev;          /* (in) */
1608     unsigned int pipe;               /* (in) endpoint + direction */
1610     int status;                      /* (return) */
1612     void *transfer_buffer;           /* (in) */
1617     u32 transfer_buffer_length;      /* (in) */
1618     u32 actual_length;               /* (return) */
1623     int interval;                   /* (modify) */
1626     void *context;                  /* (in) for complete */
1627     usb_complete_t complete;         /* (in) */
}
```

```
skbd_probe (real)

usb_fill_int_urb(k->urb, udev,
usb_rcvintpipe(udev, ep->bEndpointAddress),
k->buf, min(k->maxp, 8),
skbd_irq, k, ep->bInterval);
k->urb->transfer_dma = k->dma;
k->urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
```

1

fill: which endpoint, which buffer, how often, which function to call (usb.h 1734)

2

submit: `usb_submit_urb (urb.c 368)`; `status = -EINPROGRESS`, the URB now belongs to the controller

3

complete: the controller filled the buffer; `usbcore` calls `complete(urb)` in interrupt context

4

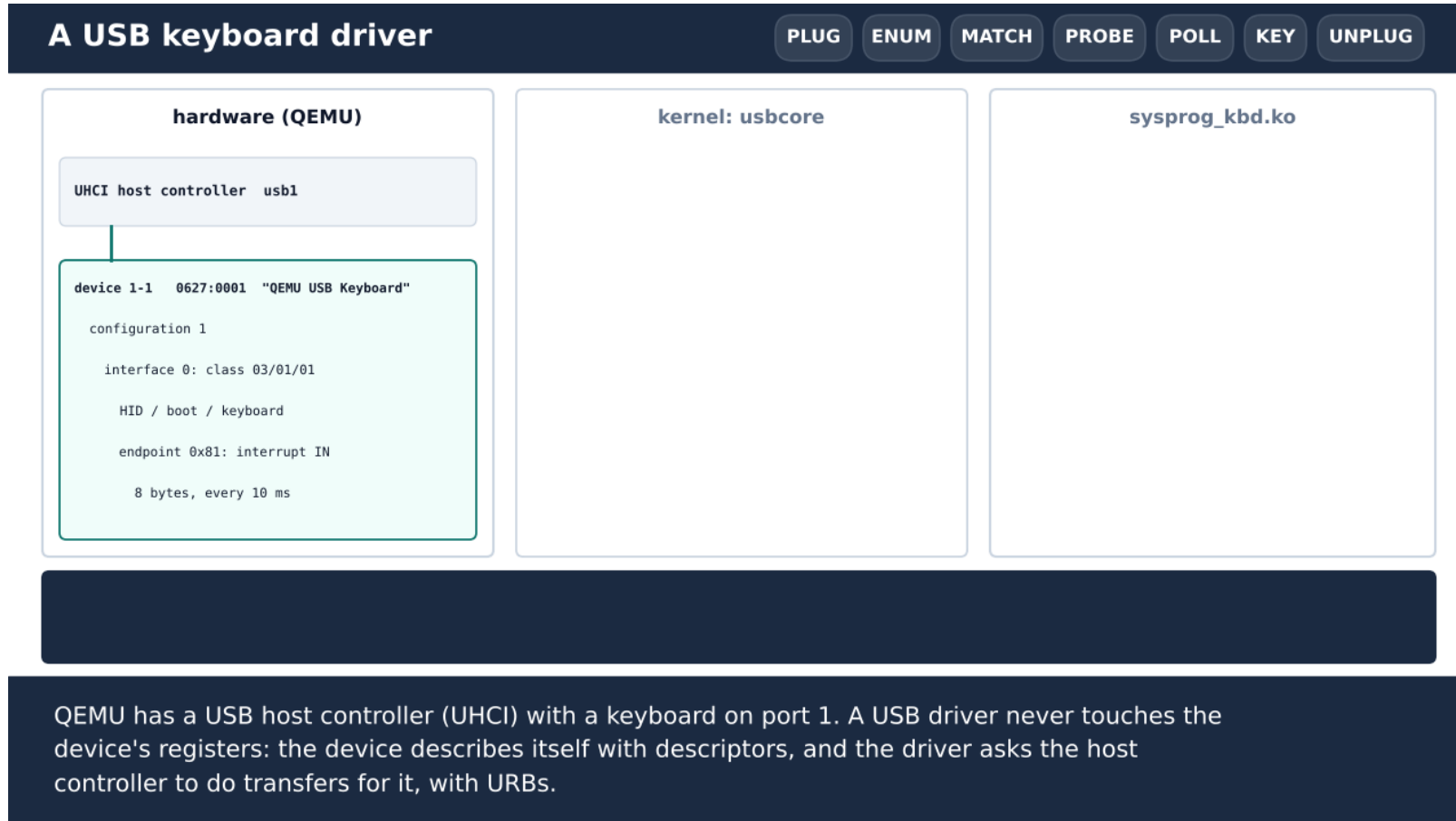
resubmit from complete, or the keyboard is never polled again

SAME AS C

A callback with a void *context

Like `pthread_create`'s `void *arg`: the kernel hands our `k` back to `skbd_irq` through `urb->context`.

Watch a USB keyboard meet its driver



Click the video to play or pause. About 1.7 minutes, no sound.

Watch for

1. Address 2, then the descriptors.
2. **usb_match_id**: 3/1/1.
3. **probe** submits one URB.
4. **NAK** every 10 ms: no CPU work.
5. Report, **skbd_irq**, resubmit.
6. **usb_kill_urb** at unplug.

skbd_irq: the completion handler, in interrupt context

● ● ● sysprog_kbd.c (real, core lines)

```
static void skbd_irq(struct urb *urb)
{
    struct skbd *k = urb->context;
    switch (urb->status) {
        case 0: break; /* data! */
        case -ECONNRESET: case -ENOENT: case -ESHUTDOWN:
            return; /* killed or unplugged */
        default: goto resubmit; /* a bad packet: try again */
    }
    dev_info(..., "report %d: %8ph key 0x%02x '%c'%s\n", ...);
resubmit:
    if (usb_submit_urb(urb, GFP_ATOMIC))
        dev_err(..., "resubmit failed\n");
}
```

allowed here?	
printk, dev_info	yes
kmalloc(..., GFP_ATOMIC)	yes: never sleeps
GFP_KERNEL, mutex_lock	no: they may sleep
copy_to_user	no: no user process here

Lecture 13 link: the same rules as the interrupt handler there: "current" is whoever was interrupted.

Why a status switch? The same function is called for success, for errors on the bus, and when the URB is killed at unplug or rmmode. Resubmitting a killed URB would fight with disconnect: return instead.

The 8-byte boot report, decoded (real)

02	00	05	00	00	00	00	00
modifiers (1)	reserved (1)	key 1 (1)	key 2 (1)	key 3 (1)	key 4 (1)	key 5 (1)	key 6 (1)

● ● ● sendkey h, i, shift-b, u → dmesg (real)

```
report 1: 00 00 0b 00 00 00 00 00 key 0x0b 'h'
report 2: 00 00 00 00 00 00 00 00 (all keys up)
report 3: 00 00 0c 00 00 00 00 00 key 0x0c 'i'
report 4: 00 00 00 00 00 00 00 00 (all keys up)
report 5: 02 00 00 00 00 00 00 00 (all keys up)
report 6: 02 00 05 00 00 00 00 00 key 0x05 'b' +shift
report 7: 02 00 00 00 00 00 00 00 (all keys up)
report 8: 00 00 00 00 00 00 00 00 (all keys up)
report 9: 00 00 18 00 00 00 00 00 key 0x18 'u'
```

bit of byte 0	key
0x01 / 0x10	left / right Ctrl
0x02 / 0x20	left / right Shift
0x04 / 0x40	left / right Alt

C link: 'a' + (u - 0x04): usage 0x0b → 'a' + 7 = 'h'.

The device reports state, not events: "these keys are down now". Pressing Shift alone is report 5; the driver must compare with the previous report to know what was pressed or released.

Unplug and plug again (real, QEMU monitor)

● ● ● skbd_disconnect (real)

```
struct skbd *k = usb_get_intfdata(intf);
usb_kill_urb(k->urb);      /* wait until not in flight */
usb_free_urb(k->urb);
usb_free_coherent(k->udev, 8, k->buf, k->dma);
dev_info(..., "disconnect after %d reports\n", k->reports);
kfree(k);
```

● ● ● (qemu) device_del kbd1 ; device_add usb-kbd,bus=uhci.0

```
usb 1-1: USB disconnect, device number 2
sysprog_kbd 1-1:1.0: disconnect after 10 reports
usb 1-2.2: new full-speed USB device number 5 using uhci_hcd
usb 1-2.2: Product: QEMU USB Keyboard
sysprog_kbd 1-2.2:1.0: probe: 0627:0001 ... class 3/1/1
sysprog_kbd 1-2.2:1.0: listening on endpoint 0x81
```

● ● ● uevents (hotplug helper log, real)

```
1188 unbind .../1-1/1-1:1.0
1189 remove .../1-1/1-1:1.0
1190 unbind .../1-1
1191 remove .../1-1
1193 change .../1-2/1-2.2
1192 add .../1-2/1-2.2
1195 bind .../1-2.2:1.0 DRIVER=sysprog_kbd
1194 add .../1-2.2:1.0
1196 bind .../1-2/1-2.2 DRIVER=usb
```

No rmmod, no insmod: the module stayed loaded; the new keyboard (now behind the hub, 1-2.2, device number 5) was probed at once. Note the numbers: **1193 before 1192**. Homework task 5.

Our driver vs the kernel's own usbkbd.c

step	sysprog_kbd.c (ours)	drivers/hid/usbhid/usbkbd.c (real lines)
which interfaces	USB_INTERFACE_INFO(3, 1, 1)	line 383: the same table
start polling	usb_fill_int_urb + usb_submit_urb	line 335: usb_fill_int_urb; submit at open
each report	skbd_irq: print it	usb_kbd_irq (line 100): input_report_key (118-133)
resubmit	usb_submit_urb(urb, GFP_ATOMIC)	line 146: the same call
user space sees	dmesg lines	/dev/input/eventN, the console, X/Wayland
unplug	usb_kill_urb, free	usb_kbd_disconnect (369): usb_kill_urb (375)

Drivers stack: usbkbd turns USB reports into **input events**; the input core and evdev (a char driver) give them to programs.

On a lab PC the keyboard is handled by usbhid + hid-generic, a more general cousin that reads the device's HID report descriptor instead of assuming the boot layout.

LAB 15.3

sysprog_kbd.c: your own USB driver

An id_table, an endpoint, a URB and its completion; keys from the QEMU monitor.

Lab 15.3 tasks, step by step

1 `cd lab15_3; make` builds `sysprog_kbd_starter.ko` and `mylsusb`

2 `USB=1 ./guest/run_guest.sh` . adds a UHCI controller, a USB keyboard and a mouse; run `mylsusb`

3 **TODO 1** `id_table`; **TODO 2** find the interrupt IN endpoint

4 **TODO 3** `usb_fill_int_urb`, `usb_set_intfdata`, `usb_submit_urb`

5 **TODO 4** decode the report and resubmit with `GFP_ATOMIC`

6 **Ctrl-A then C**: QEMU monitor. `sendkey h`, **Ctrl-A C** back, `dmesg | tail`

● ● ● the starter, before **TODO 1** (real)

```
$ insmod sysprog_kbd_starter.ko; echo status $?
status 0
$ dmesg | tail -2
# nothing: no probe, the id_table is empty
$ ls /sys/bus/usb/drivers
hub sysprog_kbd usb usbfs
```

● ● ● the finished lab (real)

```
(qemu) sendkey h
(qemu) sendkey shift-b
$ dmesg | grep report
usb 1-1: report 1: 00 00 0b 00 ... key 0x0b 'h'
usb 1-1: report 2: 00 00 00 00 ... (all keys up)
usb 1-1: report 3: 02 00 00 00 ... (all keys up)
usb 1-1: report 4: 02 00 05 00 ... key 0x05 'b' +shift
```

Checkpoint: test yourself

Q1

What does endpoint address 0x81 tell you?

→ IN (bit 7: device → host), endpoint number 1

Q2

Drivers bind to devices or to interfaces?

→ interfaces: 1-1:1.0, not 1-1

Q3

Who sends the keyboard's data when a key is pressed?

→ nobody by itself: the host controller polls every `Interval ms`

Q4

Why must `skbd_irq` resubmit the URB?

→ a URB completes once; without it, polling stops

Q5

Why `usb_kill_urb` before `kfree` in `disconnect`?

→ the completion could still run and use freed memory

Key ideas from today



A USB device is a descriptor tree; drivers bind to interfaces, data flows through endpoints.



Enumeration adds devices to the usb bus; the `id_table` and `MODALIAS` pick (and load) the driver.



probe fills and submits a URB; the completion runs in interrupt context and resubmits it.



disconnect kills the URB before freeing anything; hotplug simply repeats probe and disconnect.

Homework 15.3 (the last one)

1. Mouse

Extend the driver to the boot mouse (protocol 2): 4-byte reports of buttons, dx, dy. Decode the real report "00 fd 00 00". Why is dx a signed char?

2. MODALIAS

Decode the keyboard's and the mouse's MODALIAS field by field. Explain the real "write error: No such device" when binding the mouse to sysprog_kbd.

3. GFP_ATOMIC

skbd_irq resubmits with GFP_ATOMIC, probe with GFP_KERNEL. Explain both choices with Lecture 13's rules.

4. No usb_kill_urb

What could happen if disconnect freed k and the URB without usb_kill_urb first? Give a concrete order of events.

5. 1193 before 1192

The hotplug log shows 1193 change before 1192 add. Explain with lib/kobject_uevent.c line 618. Why does udev not have this problem?

WRAP-UP

The whole course on one page

From a SIC instruction to a USB keyboard report.

What we built, lecture by lecture

part	lectures	the question it answered	you built
machines	1, 2	What does a CPU actually run?	C probes (sizes, endianness), first nasm programs
translators	3.1–3.3, 4	How does text become machine code?	a two-pass assembler, a macro processor
linking, loading	5.1, 5.2, 6	How do pieces become one running program?	a loader, an ELF reader
the kernel	7.1, 7.2, 8.1, 8.2, 9	Who runs the program, and how does it ask for help?	kinfo, a tiny init, a shell, a program without libc
processes talk	10.1, 10.2	How do programs cooperate and get interrupted?	a shell with pipes, mytimeout with signals
memory, files	11.1, 11.2, 12.1, 12.2	Where do bytes live, in RAM and on disk?	mymalloc, vtop, myls, ext2cat
hardware	13, 14.1, 14.2	How does the kernel meet the hardware?	irqtop, a /proc module, a platform driver
drivers	15.1–15.3	How does a device become a file, a disk, a key?	sysbuf, sbd, sysprog_kbd

One key press, through everything we studied

- 1 the host controller polls, the keyboard answers: **an interrupt** on IRQ 11 (Lecture 13)
- 2 the completion runs **driver code**: a module the kernel linked at run time (14.1, like 5.1's loader)
- 3 the driver found its device on a **bus** by matching an `id_table` (14.2, 15.3)
- 4 the key reaches a program through a **char device** `read()` (15.1, 12.1's file operations)
- 5 that program is a **process** whose system calls (9) run kernel code; its ELF (6) was assembled (3) and linked (5)
- 6 and every instruction on that path is **fetch, decode, execute** (Lecture 1)

System programming is the software that makes the machine usable by other software: translators, loaders, the kernel, drivers.

The habit to keep: never guess what the system does: **measure it**. `strace`, `ftrace`, `/proc`, `/sys`, `objdump` and `readelf` answered every question in this course with real output.

Where to go from here

END OF CSE 4105

Thank you.

Final exam: every lecture's key-ideas slide, checkpoint and homework are the best review. Bring questions to office hours.

Read

LDD3 (free online) Ch. 13–14; Love, **LKD** Ch. 13–17; Bryant & O'Hallaron, **CS:APP** Ch. 7–10 again, now with the kernel's view.

Try

Port `sysprog_kbd` to report keys through the input subsystem (`input_allocate_device`, `input_report_key`), like `usbkbd.c`.

Explore

Read one real driver end to end: `drivers/block/brd.c` (RAM disk), `drivers/char/mem.c` (`/dev/null`, `/dev/zero`), `drivers/hid/usbhid/usbkbd.c`.

Contribute

kernelnewbies.org and the kernel's Documentation/process/: how patches are written, tested and sent. QEMU lets you test anything safely, as in our labs.