Capítulo 26 — Input en Linux: Teclado, Mouse y Joystick con evdev, libudev y libinput

> *Capítulo avanzado, pero **escrito para que se entienda de verdad**. Se apoya en file descriptors (`22-file-descriptors`), punteros (`07`), memoria dinámica (`09`) y structs (`08`).*
>
> *Este capítulo te lleva de "leer el teclado con `getchar()` " a leer eventos crudos de `/dev/input/eventN` directamente desde el kernel, enumerar dispositivos con libudev, y abstraer todo en una librería que use libinput para unificar teclado, mouse y joystick.*

Cómo leer este capítulo

Hasta ahora leste el teclado con `getchar()` o `scanf()` . Eso es lo que hace la **capa más alta** del sistema: decile al usuario "acá escribiste 'A'", punto. Pero en Linux, el teclado no es magia: es un archivo `/dev/input/eventN` por cada dispositivo, y dentro hay eventos crudos con código de tecla, tipo de evento, timestamp y valor.

Este capítulo va de **cero absoluto** (qué es un evento evdev) hasta una **librería completa** que capture teclado, mouse y joystick con una API uniforme: `on_key_down(keycode)` , `on_mouse_button(button, pressed)` , `on_joystick_axis(axis, value)` .

Está dividido en **ocho partes**. Después de cada concepto hay una o más prácticas —uno o más ejercicios— para que los uses ya mismo. **No las saltee**: son la mitad del capítulo.

Parte	De qué habla	La idea en una frase
1	¿Cómo llega una pulsación al teclado?	El kernel genera un evento → lo escribe en `/dev/input/eventN` → tu programa lo lee como bytes crudos.
2	El formato de `input_event`	Cada evento es un struct con `timeval`, `type`, `code`, `value`: leé eventos del kernel sin intermediarios.
3	Leer el teclado crudo con `ioctl`	`EVIOCGNAME` y `EVIOCGBIT`: abrí, configurá y obtené el nombre del dispositivo conectado a tu teclado.
4	Enumerar dispositivos con libudev (método fácil)	No busques a mano: libudev lee udev y te dice qué fd es qué dispositivo, su modalias, su path completo.
5	El protocolo de eventos evdev	Tipos `EV_KEY`, `EV_REL`, `EV_ABS`: entendé la diferencia entre botón pulsado, movimiento relativo y eje analógico.
6	Mouse: movimiento relativo y botones	`EV_REL` con `REL_X`, `REL_Y`, `REL_WHEEL` + `EV_KEY` con `BTN_LEFT`, `BTN_RIGHT`: leé el mouse crudo.
7	Joystick: ejes absolutos y botones	`EV_ABS` con `ABS_X`, `ABS_Y`, `ABS_Z`, `ABS_RZ`, `ABS_THROTTLE`: mapeá posiciones analógicas de un gamepad crudo.
8	Ejercicio jefe: capturador universal de input	Librería unificada que abstraiga teclado, mouse y joystick en una API con callbacks por evento.

> ***Regla de oro del libro:*** *el capítulo explica y muestra* ***cómo funciona*** *cada syscall/ioctl/función, pero* ***los ejercicios no traen la solución****. El enunciado te da el problema; la respuesta la escribís vos.*

PARTE 1 — ¿Cómo llega una pulsación al teclado?

Antes de tocar código, entendamos qué pasa cuando presionás una tecla. Porque lo que sucede entre tu dedo y la pantalla es **más largo** de lo que creés — y más interesante todavía.

1.1 El camino de un "A" desde tu dedo hasta la terminal

Imaginate que tu teclado es una **fábrica de eventos**. Cada vez que presionás una tecla, algo termina en pantalla. Pero ¿qué pasa en el medio? Acá está el recorrido completo:

```
dedo → matriz de switches (hardware del teclado) → controlador USB/HID → kernel input subsystem → /dev/input/eventN → tu programa
```

Cada flecha es un **paso**. Uno detrás del otro. Si cualquiera falla, no llegás a la pantalla.

La primera parada es el **hardware del teclado**: una matriz de switches (mecánicos o de goma) que cierra circuitos eléctricos cuando presionás una tecla. El controlador del teclado lee esa matriz (una vez por cada milisegundo más o menos) y genera un "scancode": un número que identifica qué switch se cerró, sin importar qué letra imprima.

El scancode viaja por USB (o Bluetooth, o PS/2 legacy) hasta el **kernel**, que es donde empieza la magia de Linux: el **input subsystem**.

1.2 El input subsystem: el traductor universal de Linux

El kernel input subsystem es el subsistema del kernel que **normaliza todos los dispositivos de entrada en un formato uniforme**, sin importar si es USB, Bluetooth, PS/2 o serial:

> ***Todo dispositivo de entrada se convierte en eventos.*** *No hay diferencia entre "presionar tecla" y "mover mouse": ambos son secuencias de structs* `input_event` *escritos en file descriptors especiales. Esos archivos viven en* `/dev/input/` *.*

El input subsystem tiene tres capas que trabajan en cadena:

Capa	Qué hace	Ejemplo
HID driver	Traduce protocolos (USB HID, PS/2, Bluetooth HOGP) a eventos genéricos del kernel	Tu teclado USB envía reports HID → el driver los traduce a `EV_KEY`
Input core	Normaliza eventos y los pone en `/dev/input/eventN`	Unifica teclas de 104 teclas, teclados multimedia, laptops con touchpad
Userspace	Tu programa lee los eventos desde el fd	`read(event_fd, &buf, sizeof(buf))`

La capa HID driver es la que conoce el protocolo específico de cada dispositivo. Un teclado USB usa el protocolo "HID" (Human Interface Device), un joystick Bluetooth usa "HOGP" (HID Over GATT Profile), un touchpad PS/2 usa su propio protocolo. **El driver traduce todo a eventos genéricos** que el input core puede manejar uniformemente.

El input core es el cerebro: recibe los eventos del driver, les asigna un tipo (`EV_KEY` , `EV_REL` , `EV_ABS`) y los escribe en el fd correspondiente de `/dev/input/` .

Y vos estás en la última capa: leyendo esos eventos con `read()` .

> ***La analogía del traductor:*** *imaginá que el HID driver es un traductor en una conferencia internacional. Recibe palabras en 20 idiomas distintos (USB, Bluetooth, PS/2) y las traduce a una lengua común (eventos genéricos). El input core es el sistema de subtítulos que muestra esas palabras traducidas en la pantalla (`/dev/input/eventN`). Y vos sos el público: no te importa qué idioma hablaba cada persona, solo leés los subtítulos.*

1.3 Los archivos `/dev/input/`

```
$ ls /dev/input/
by-id/  by-path/  event0  event1  event10  js0  mice  mouse0
```

Cada `eventN` es un **file descriptor de caracteres** que el kernel usa para enviar eventos de entrada. Pero no todos los eventos van al mismo archivo: el kernel decide qué dispositivo va a qué `eventN` basándose en el tipo de hardware.

Path	Qué contiene	Ejemplo típico
`/dev/input/event0`	Eventos genéricos (teclado, mouse, etc.)	Teclado USB principal
`/dev/input/event1`	Más eventos genéricos	Mouse o touchpad
`/dev/input/js0`	Joysticks antiguos (legacy)	Gamepad antiguo
`/dev/input/mice`	Mouse PS/2 legacy (agregado de varios ratones)	Laptop con trackpad
`/dev/input/by-id/`	Links simbólicos estables por ID físico	`usb-` … `-event-kbd`
`/dev/input/by-path/`	Links simbólicos estables por path físico	`pci-0000:00:14.0-usb-` `event`

> ***No hardcodees*** `/dev/input/event0` ***.*** *Cambia según el orden de conexión USB, qué dispositivos estaban conectados al arrancar, si usás Bluetooth o no, etc. Siempre buscá dinámicamente.*

1.3.1 Por qué los links por-id y by-path existen

`/dev/input/event0` es **inestable**: si desconectás el teclado y lo volvé a conectar, puede pasar a ser `event5`. Si conectás un segundo teclado, todos los números se desplazan.

Los links en `by-id/` y `by-path/` son **estables**: apuntan al mismo dispositivo físico siempre, sin importar qué número de event tenga. Son útiles cuando necesitás abrir un dispositivo específico (sin ambigüedad).

```
$ ls -la /dev/input/by-id/
total 0
lrwxrwxrwx 1 root root  10 jun 15 10:30 usb-Logitech_G502_HERO-event-mouse -> ../event7
```

Si desconectás y reconectás el mouse, el link sigue apuntando al mismo hardware — aunque el event subyacente cambie de número.

> ***Para este capítulo*** *no necesitamos links estables: vamos a enumerar todos los dispositivos activos y procesarlos. Pero si escribís un daemon o un servicio que arranca con el sistema, usá* `by-id/` *o* `by-path/` *.*

1.4 Lo que te llevás de esta parte

- **El input subsystem normaliza todo.** Sin importar el protocolo (USB, Bluetooth, PS/2), el kernel convierte cada dispositivo en eventos genéricos.
- **Los eventos viven en** `/dev/input/eventN` **.** Cada dispositivo tiene uno o más event nodes.
- `event0` **es inestable.** Usá `by-id/`, `by-path/`, o mejor aún: libudev para enumerar dinámicamente.

Práctica rápida de la Parte 1

Ejercicio 26.1 — Listar dispositivos de input

Conceptos · `opendir` , `readdir` , `stat` , `/dev/input/event*`

Leé el contenido de `/dev/input/` y listá todos los archivos que empiecen con `event` . Para cada uno, mostrá:

- El nombre del archivo (ej: `event5`)
- Si es un dispositivo de caracteres (`c` para character device)
- Su tamaño (siempre 0, porque son devices — no tiene dato útil, es una trampa común)

Salida esperada:

```
Dispositivos en /dev/input/:
  event0   [char dev]
  event1   [char dev]
  event2   [char dev]
  mice     [regular file]
  mouse0   [char dev]
  ...
```

Criterios de aceptación:

- ☐ Recorrés `/dev/input/` con `opendir` / `readdir` , no con `system("ls")` .
- ☐ Usás `stat()` para verificar si es character device (`S_ISCHR`).
- ☐ Mostrás todos los entries del directorio, no solo `eventN` .

PARTE 2 — El formato de `input_event`

2.1 La estructura más importante del input en Linux

Todo evento de entrada viaja como un `struct input_event`:

```
#include <linux/input.h>

struct timeval {
    time_t       tv_sec;   // segundos desde epoch (Unix timestamp)
    suseconds_t  tv_usec;  // microsegundos dentro del segundo
};

struct input_event {
    struct timeval time;   // cuándo ocurrió el evento (con microsegundos de precisión)
    __uint16_t type;       // qué tipo de evento es: tecla, movimiento, sincronización...
    __uint16_t code;       // cuál código dentro del tipo: qué tecla, qué botón, qué eje
    __s32        value;    // el valor: 0/1 para pulsado/soltado, delta para movimiento,
posición para ejes
};
```

Un solo `read()` lee **un evento completo**. Siempre. No hay eventos incompletos ni fragmentados:

```
struct input_event ev;
while (read(fd, &ev, sizeof(ev)) == sizeof(ev)) {
    // procesar evento
}
```

> ***¿Por qué `timeval` y no `clock_gettime`?*** *Porque el kernel usa un reloj interno para timestampear cada evento en el momento exacto en que ocurre. No es un reloj arbitrario: es el tiempo del sistema, con microsegundos de precisión. Podés convertirlo a cualquier formato después.*

2.2 Los tres campos decodificados

Cada evento tiene exactamente **cuatro campos**, pero solo `type`, `code` y `value` son los que te importan para interpretar qué pasó:

Campo	Tipo	Qué significa	Ejemplo de valor
`time`	`struct timeval`	Cuándo ocurrió el evento	`tv_sec=1700000000, tv_usec=123456`
`type`	`uint16_t`	Categoría del evento: tecla, movimiento relativo, eje absoluto	`EV_KEY` (1), `EV_REL` (3), `EV_ABS` (3)
`code`	`uint16_t`	Identificador dentro de la categoría	`KEY_A`, `REL_X`, `ABS_X`
`value`	`int32_t`	El dato concreto del evento	`1` (presionado), `-5` (delta), `32768` (posición)

> ***La regla de oro de `input_event`:*** *el significado de `value` depende completamente de `type` y `code`. Un `value=1` significa "presionado" para `EV_KEY`, pero tiene otro significado para `EV_ABS` (valor absoluto del eje) y no existe para `EV_SYN`.* ***Siempre mirá los tres campos juntos.***

2.3 Valores según el tipo de evento

EV_KEY: teclas y botones

`code`	`value = 0`	`value = 1`	`value = 2`
`KEY_A`, `KEY_ENTER`, `BTN_LEFT`...	Soltado (release)	Presionado (press)	Repeat (mantener presionado)

Para `EV_KEY`, el valor siempre es 0, 1 o 2:

```
tecla suelta      → value=0   (release)
tecla pulsada     → value=1   (press)
tecla manteniendo → value=2   (repeat, generado automáticamente por el kernel tras ~500ms)
```

El "repeat" (`value=2`) es un evento del **kernel**, no del hardware. El kernel detecta que la tecla sigue presionada y genera eventos de repeat automáticamente. Podés desactivarlos con `ioctl` , pero normalmente los ignorás (ya sabés que la tecla sigue presionada por el press anterior).

EV_REL: movimiento relativo (mouse)

code	value	Significado
`REL_X`	`+5`	Mouse se movió 5 unidades a la derecha
`REL_Y`	`-3`	Mouse se movió 3 unidades arriba
`REL_WHEEL`	`+1`	Scroll: una línea hacia abajo (alejado)
`REL_WHEEL`	`-1`	Scroll: una línea hacia arriba (hacia el usuario)
`REL_HWHEEL`	`+2`	Scroll horizontal: 2 pasos a la derecha

Para `EV_REL` , el valor es siempre un **delta**: cuánto se movió desde el último evento, sin ninguna posición absoluta de referencia. Un mouse USB reporta deltas típicamente entre -128 y +127 por evento. Si movés el mouse rápido, podés recibir deltas grandes o múltiples eventos REL_X seguidos.

> ***Los deltas no suman para dar una posición absoluta.*** *Si querés la posición del cursor en pantalla, tenés que acumularlos vos:* `cursor_x += delta_x` , `cursor_y += delta_y` . *Pero cuidado: si el mouse se mueve entre frames (no leés lo suficientemente rápido), podés recibir decenas de eventos REL acumulados.*

EV_ABS: posición absoluta (joystick, touchpad)

code	value	Significado
`ABS_X`	`32768`	Eje X en el centro (rango típico: 0–65535 o -32768 a +32767)
`ABS_Y`	`100`	Eje Y muy arriba
`ABS_Z`	`65535`	Trigger izquierdo presionado a fondo (si el rango es 0–65535)

Para `EV_ABS` , el valor es una **posición absoluta** dentro de un rango definido por el dispositivo. Si un joystick reporta `ABS_X = 32768` , significa "en el centro", sin importar si el usuario movió el stick antes o no. Es como tener un termómetro que te dice "estamos a 25°C", no "la temperatura subió 2 grados".

> ***Ojo con los rangos de ABS:*** *algunos joysticks reportan* `-32768` *a* `+32767`*, otros* `0` *a* `65535`*, y hay devices baratos que reportan rangos completamente distintos (ej:* `1000` *a* `64000`*).* ***Nunca asumas el rango*** *— consultá con* `EVIOCGABS` *(llegamos a esa parte).*

EV_SYN: sincronización

`code`	`value`	Significado
`SYN_REPORT`	`0`	Todos los eventos anteriores forman un frame completo
`SYN_CONFIG`	`0`	La configuración del dispositivo cambió
`SYN_MT_REPORT`	`0`	Fin de un reporte multitouch (para pantallas táctiles)

`EV_SYN` es el "punto y coma" de los eventos. El kernel agrupa eventos relacionados con un SYN_REPORT:

```
EV_KEY + KEY_A      → value=1     (presionaste A)
EV_SYN + SYN_REPORT → value=0    (este frame está completo)
EV_KEY + KEY_A      → value=0     (solto A)
EV_SYN + SYN_REPORT → value=0    (este frame también)
```

En la práctica, **no necesitás procesar SYN_REPORT** a menos que estés construyendo algo sensible a frames completos (un compositor de ventana, un juego competitivo). Para lectura simple, ignoralo.

> ***¿Por qué existen los SYN?*** *Porque el kernel puede recibir múltiples eventos del mismo dispositivo en un solo* `read()`*. Sin SYN, no sabrías si dos eventos están relacionados o son independientes. SYN_REPORT dice: "todo lo que leíste antes de este byte es un frame atómico".*

2.4 Decodificar un evento paso a paso

Acá tenés un ejemplo mínimo que imprime cada evento en texto legible:

```c
#include <stdio.h>
#include <linux/input.h>
#include <fcntl.h>
#include <unistd.h>

void print_event(struct input_event *ev) {
    printf("time=%ld.%06d  type=%-4u  code=%-5u  value=%-6d\n",
           ev->tv_sec, ev->tv_usec / 1000,       // timestamp en ms
           ev->type,                               // tipo numérico
           ev->code,                               // código numérico
           ev->value);                             // valor
}

int main(void) {
    int fd = open("/dev/input/event3", O_RDONLY | O_NONBLOCK);
    if (fd < 0) {
        perror("open");
        return 1;
    }

    struct input_event ev;
    while (read(fd, &ev, sizeof(ev)) == sizeof(ev)) {
        print_event(&ev);
    }

    close(fd);
    return 0;
}
```

¿Por qué `O_NONBLOCK`? *Porque si el dispositivo no genera eventos (no tocás nada), `read()` se bloquearía indefinidamente. Con `O_NONBLOCK`, devuelve `-1` con `errno=EAGAIN` cuando no hay datos — así podés hacer otras cosas o usar `poll()` / `select()` para esperar de forma eficiente.*

2.5 Lo que te llevás de esta parte

- **Todo evento es un `input_event`:** struct con timestamp, tipo, código y valor.
- **`read()` devuelve un evento completo siempre.** No hay eventos parciales.
- **El significado de `value` depende de `type`.** 0/1 para teclas, delta para movimiento relativo, posición absoluta para ejes.
- **`EV_SYN` separa frames.** No lo procesás directamente, pero sabé que existe.

Práctica rápida de la Parte 2

Ejercicio 26.2 — Lectura cruda de eventos de teclado

Conceptos · `open`, `read`, `struct input_event`, `O_NONBLOCK`

Encontrá el fd del teclado (pista: tiene `EV_KEY` en su bitmask con `EVIOCGBIT(fd, EV_MAX)`), abrilo **en modo no bloqueante**, y leé 20 eventos mostrados:

```
time=1700000000.123456 type=1 code=30 value=1  → KEY_A pressed
time=1700000000.134567 type=1 code=30 value=0  → KEY_A released
```

Criterios de aceptación:

- ☐ Abrís el fd con `O_NONBLOCK`.
- ☐ Mostrás solo los primeros 20 eventos de tipo `EV_KEY` (type == EV_KEY).
- ☐ Traducís `value`: 0 = "released", 1 = "pressed", 2 = "repeat".

PARTE 3 — Leer el teclado crudo con `ioctl`

3.1 ioctl del input subsystem

Los `ioctl` del subsistema de entrada se usan con el fd de un dispositivo `/dev/input/eventN`. Son los comandos más útiles:

ioctl	Qué hace	Ejemplo de uso
`EVIOCGNAME(sizeof(name))`	Obtiene el nombre descriptivo del dispositivo	"AT Translated Set 2 Keyboard"
`EVIOCGVERSION(version)`	Obtiene la versión del driver	Major/minor/patch
`EVIOCGBIT(ev, size, data)`	Obtén el bitmask de eventos soportados	¿Qué tipos reporta este device?
`EVIOCGID(fd, &id)`	Info del dispositivo (bus/vendor/product/version)	USB vendor/product ID
`EVIOCGABS(code, &abs)`	Obtiene parámetros de un eje absoluto	min/max/resolution/fuzz/flat
`EVIOCSREP(delay, rate)`	Configura delay y rate de repeat de teclas	Cuánto esperar antes de repetir
`EVIOCGRAB(fd, 1)`	Grab: nadie más recibe eventos mientras tengas el fd abierto	Secuestrar el dispositivo

> ***`EVIOCGBIT(ev, size, data)`:*** *este es el comando más importante para descubrir qué hace un dispositivo. Le pedís al kernel "dame los bits de tipos de evento soportados por este fd" y el kernel te devuelve un bitmask donde cada bit corresponde a un tipo (`EV_KEY`, `EV_REL`, etc.).*

3.1.1 EVIOCGNAME: saber qué device es cuál

```c
char name[256] = {0};
ioctl(fd, EVIOCGNAME(sizeof(name)), name);
printf("Device name: %s\n", name);
// Output: "AT Translated Set 2 Keyboard"
//         "Logitech G502 HERO Gaming Mouse"
//         "Xbox Wireless Controller"
```

`EVIOCGNAME` te devuelve una cadena legible por humanos que identifica el dispositivo. Es mucho más útil que adivinar cuál es el teclado mirando `/dev/input/eventN`.

3.1.2 EVIOCGBIT: descubrí qué hace un dispositivo

Este es el comando que usás para **descubrir qué tipo de eventos reporta cada dispositivo**. Te devuelve un bitmask donde cada bit activo significa que el dispositivo genera ese tipo de evento:

```c
#include <linux/input.h>
#include <sys/ioctl.h>

unsigned long types[BITS_TO_LONGS(EV_MAX)];
ioctl(fd, EVIOCGBIT(0, sizeof(types)), types);

// test_bit verifica si un bit está seteado en el bitmask
int has_keys  = test_bit(EV_KEY, types);
int has_rel   = test_bit(EV_REL, types);
int has_abs   = test_bit(EV_ABS, types);
int has_msc   = test_bit(EV_MSC, types);

printf("EV_KEY: %s  EV_REL: %s  EV_ABS: %s  EV_MSC: %s\n",
       has_keys ? "YES" : "no",
       has_rel  ? "YES" : "no",
       has_abs  ? "YES" : "no",
       has_msc  ? "YES" : "no");
```

`BITS_TO_LONGS(n)` *es una macro que calcula cuántos* `unsigned long` *necesitás para almacenar* `n` *bits. Es equivalente a* `(n + 31) / 32`*. La usá siempre con* `EVIOCGBIT`*.*

`test_bit(bit, array)` *verifica si el bit en la posición* `bit` *está activo (seteado a 1). Es una macro que hace operaciones bitwise sobre el array devuelto por* `EVIOCGBIT`*.*

3.2 Clasificar dispositivos: keyboard vs mouse vs joystick

Con `EVIOCGBIT`, podés distinguir qué tipo de dispositivo es cada uno:

```
InputDeviceType classify_device(int fd) {
    unsigned long types[BITS_TO_LONGS(EV_MAX)];
    ioctl(fd, EVIOCGBIT(0, sizeof(types)), types);

    int has_keys  = test_bit(EV_KEY, types);
    int has_rel   = test_bit(EV_REL, types);
    int has_abs   = test_bit(EV_ABS, types);

    // Reglas de clasificación:
    if (has_keys && !has_rel && !has_abs) {
        return DEVICE_KEYBOARD;     // solo teclas → keyboard
    }
    else if (has_keys && has_rel && !has_abs) {
        return DEVICE_MOUSE;        // botones + movimiento relativo, sin ejes absolutos
→ mouse
    }
    else if (has_keys && has_rel && has_abs) {
        return DEVICE_GAMEPAD;      // tiene botones Y ejes absolutos → gamepad/joystick
    }
    else if (!has_keys && has_abs) {
        return DEVICE_JOYSTICK;     // solo ejes absolutos (ej: un acelerador
independiente) → joystick
    }
    else {
        return DEVICE_UNKNOWN;      // no clasificado con las reglas conocidas
    }
}
```

Las reglas de clasificación no son perfectas. *Un touchpad tiene* `EV_KEY` *(click),* `EV_REL` *(movimiento), y a veces* `EV_ABS` *(gestos multitouch). Lo importante es entender el patrón:* ***mirá qué bits están activos en el bitmask*** *y decidí basado en esa combinación.*

3.3 El modo grab: secuestrar un dispositivo

`EVIOCGRAB(fd, 1)` hace que **ningún otro programa** reciba eventos de este fd mientras tu proceso lo tenga abierto con el grab activo. Es lo que usan herramientas legítimas como `showkeys`, `xdotool`, o juegos en pantalla completa — pero también es la base de los keyloggers maliciosos.

```
// Grabar teclado: solo mi programa recibe eventos
if (ioctl(fd, EVIOCGRAB, 1) < 0) {
    perror("EVIOCGRAB failed");
    // Probablemente necesitás root o pertenecer al grupo 'input'
}

// ... leer eventos sin interferencia de otros programas ...

// Soltar teclado: otros programas vuelven a recibir eventos
ioctl(fd, EVIOCGRAB, 0);
```

> ***Necesitás root (o permiso del grupo `input`) para hacer grab.*** *Sin ellos, podés leer pero no excluir a otros programas. Es una medida de seguridad: nadie debería poder secuestrar tu teclado sin tu permiso explícito.*

3.4 EVIOCGABS: parámetros de ejes absolutos

Cada eje absoluto tiene un **rango** y **parámetros de calibración**. Podés consultarlo con `EVIOCGABS`:

```
struct input_absinfo abs;
ioctl(fd, EVIOCGABS(ABS_X), &abs);

printf("Eje ABS_X:\n");
printf("  min:     %d\n", abs.minimum);      // valor mínimo del eje
printf("  max:     %d\n", abs.maximum);      // valor máximo del eje
printf("  fuzz:    %d\n", abs.fuzz);         // ruido aceptable (ignorar cambios menores)
printf("  flat:    %d\n", abs.flat);          // zona muerta central (deadzone)
printf("  resolution %d\n", abs.resolution);  // unidades por milímetro (o grado, etc.)
```

> ***No asumas que todos los ejes van de `-32768` a `+32767`.*** *Joysticks baratos, pads de guitarra, volantes — cada uno puede tener rangos distintos. Siempre consultá con `EVIOCGABS`.*

3.5 Lo que te llevás de esta parte

- **ioctl son la API de bajo nivel del kernel para dispositivos.** Con `EVIOCGBIT` descubrí qué hace un dispositivo, con `EVIOCGNAME` su nombre descriptivo, con `EVIOCGABS` los parámetros de ejes.
- **La clasificación se basa en bits:** solo `EV_KEY` = keyboard, `EV_KEY + EV_REL` sin `EV_ABS` = mouse, `EV_ABS` presente = joystick/gamepad.
- **EVIOCGRAB secuestra el dispositivo.** Nadie más recibe eventos mientras tengas el grab activo. Requiere root o grupo `input`.

Práctica rápida de la Parte 3

Ejercicio 26.3 — Descubrir tipos de dispositivos del input

Conceptos · `ioctl`, `EVIOCGBIT`, bitmask con `test_bit`, `open`

Abrí cada `/dev/input/eventN` y determiná qué tipo de dispositivo es (teclado, mouse, touchpad, joystick) usando los bitmasks de eventos soportados. Mostrá:

```
event0  → keyboard   ("AT Translated Set 2 Keyboard")
event1  → mouse      ("SynPS/2 Synaptics TouchPad")
event3  → gamepad    ("Xbox Wireless Controller")
```

Criterios de aceptación:

- ☐ Usás `EVIOCGBIT` para cada fd.
- ☐ Clasificás correctamente: keyboard (solo EV_KEY), mouse (EV_REL + EV_KEY sin EV_ABS), joystick/gamepad (EV_ABS presente).
- ☐ Mostrás el nombre del dispositivo con `EVIOCGNAME`.
- ☐ Manejás errores si un fd no se puede abrir (permisos, dispositivo eliminado entre tanto).

PARTE 4 — Enumerar dispositivos con libudev (método fácil)

4.1 ¿Por qué usar libudev en vez de leer `/dev/input/` ?

Leer `/dev/input/eventN` a mano tiene problemas serios:

- No sabés **qué dispositivo es** sin abrirlo y hacer `ioctl` .
- No sabés **cuándo se conectó o desconectó** un dispositivo nuevo (hotplug).
- No tenés acceso a **propiedades del dispositivo**: vendor ID, product ID, modalias, path físico.
- No podés saber si el usuario tiene permiso para leerlo sin intentarlo y ver si falla.

libudev resuelve todo eso: lee el sysfs y udev database para darte metadatos de cada dispositivo **antes de abrirlo**. Es la capa de abstracción correcta sobre `/dev/input/` .

> ***La analogía del directorio telefónico:*** *leer* `/dev/input/` *es como buscar números de teléfono en un listín sin nombres, solo números. libudev es el directorio con nombres, direcciones y categorías: "Logitech G502 → mouse gaming → vendor 0x046d". Sabés qué estás buscando antes de marcar.*

4.2 El flujo básico de libudev

```c
#include <libudev.h>

// 1. Crear el contexto udev (una sola vez)
struct udev *udev = udev_new();
if (!udev) {
    perror("udev_new failed");
    return 1;
}

// 2. Crear un enumerador (buscador)
struct udev_enumerate *enumerate = udev_enumerate_new(udev);

// 3. Agregar filtros: solo dispositivos del subsistema "input"
udev_enumerate_add_match_subsystem(enumerate, "input");

// 4. Escanear y obtener la lista
udev_enumerate_scan_devices(enumerate);

// 5. Recorrer los resultados
struct udev_list_entry *entry;
udev_list_entry_foreach(entry, udev_enumerate_get_list_entry(enumerate)) {
    const char *syspath = udev_list_entry_get_name(entry);

    // Para cada entry, crear un device object con todos sus metadatos.
    struct udev_device *dev = udev_device_new_from_syspath(udev, syspath);

    // Obtener propiedades del dispositivo (si están disponibles):
    const char *devname  = udev_device_get_devnode(dev);       // /dev/input/eventX
    const char *name     = udev_device_get_property_value(dev, "ID_INPUT");
    const char *vendor   = udev_device_get_property_value(dev, "ID_VENDOR_id");
    const char *product  = udev_device_get_property_value(dev, "ID_MODEL_id");
    const char *modalias = udev_device_get_property_value(dev, "modalias");

    printf("Device: %s (%s) vendor=%s product=%s\n",
           devname, modalias ? modalias : "unknown",
           vendor ? vendor : "?", product ? product : "?");

    // Liberar el device object (es referencia-contada)
    udev_device_unref(dev);
}

// 6. Liberar recursos
udev_enumerate_unref(enumerate);
udev_unref(udev);
```

> ***Siempre liberá los objetos.*** *Cada* `udev_device_new_from_syspath()` *devuelve un objeto con referencia contada. Llamá a* `udev_device_unref(dev)` *cuando termines, y* `udev_unref(udev)` *al final del programa para liberar el contexto global.*

4.3 Propiedades clave de libudev

Propiedad	Qué dice	Ejemplo
`ID_INPUT`	"1" si es un dispositivo de entrada	Todos los `/dev/input/eventN`
`ID_INPUT_KEYBOARD`	"1" si tiene teclas	Teclados, laptops
`ID_INPUT_MOUSE`	"1" si es un ratón	Mouse USB, trackpad
`ID_INPUT_JOYSTICK`	"1" si es gamepad	DualSense, Logitech
`ID_INPUT_TOUCHPAD`	"1" si es touchpad	Synaptics, ELAN
`modalias`	Tipo de dispositivo (USB/PS/2/HID)	`"input:test-0G0M ... "`
`DEVNAME`	Path al device node	`/dev/input/event5`

> ***Las propiedades udev son las más confiables*** *para clasificar dispositivos. Si* `ID_INPUT_KEYBOARD` *está presente y vale "1", el dispositivo ES un teclado — no hay que adivinarlo con bits.*

4.4 Fallback: detectar tipo con ioctl (cuando udev no alcanza)

A veces udev no tiene la propiedad que necesitás: dispositivos genéricos, contenedores sin udevd corriendo, o devices antiguos. En esos casos, **caés al kernel como red de seguridad**:

```
InputDeviceType detect_by_ioctl(const char *devpath) {
    int fd = open(devpath, O_RDONLY | O_NONBLOCK);
    if (fd < 0) return DEVICE_UNKNOWN;

    unsigned long types[BITS_TO_LONGS(EV_MAX)];
    ioctl(fd, EVIOCGBIT(0, sizeof(types)), types);

    int has_keys  = test_bit(EV_KEY, types);
    int has_rel   = test_bit(EV_REL, types);
    int has_abs   = test_bit(EV_ABS, types);

    // Reglas de clasificación:
    if (has_keys && !has_rel && !has_abs) {
        close(fd);
        return DEVICE_KEYBOARD;       // solo teclas → keyboard
    }
    else if (has_keys && has_rel && !has_abs) {
        close(fd);
        return DEVICE_MOUSE;          // botones + relativo sin absoluto → mouse
    }
    else if (has_abs) {
        close(fd);
        return DEVICE_JOYSTICK;       // tiene ejes absolutos → joystick/gamepad
    }

    close(fd);
    return DEVICE_UNKNOWN;
}
```

> ***Siempre empezá por la abstracción más alta.*** *Udev es la capa fácil — chequeá sus propiedades primero. Si no clasifica, usá ioctl como fallback. Este patrón (abstracción → raw) se repite en muchos subsistemas de Linux.*

4.5 Monitorizar hotplug con libudev

libudev permite escuchar eventos en tiempo real: cuando conectás o desconectás un dispositivo:

```c
// Crear un monitor que escuche eventos del kernel udevd
struct udev_monitor *mon = udev_monitor_new_from_netlink(udev, "udev");
udev_monitor_filter_add_match_subsystem_devtype(mon, "input", NULL);
udev_monitor_enable_receiving(mon);

// El fd del monitor se puede usar con poll()/select() junto con los fds de input:
struct pollfd pfds[3];
pfds[0].fd = udev_monitor_get_fd(mon);  // hotplug events
pfds[1].fd = keyboard_fd;               // teclado
pfds[2].fd = mouse_fd;                  // mouse
// ... más fds para otros dispositivos ...

int ret = poll(pfds, num_fds, -1);  // espera indefinida
if (pfds[0].revents & POLLIN) {
    // Nuevo dispositivo conectado o desconectado!
    struct udev_device *hotplug_dev = udev_monitor_receive_device(mon);
    // ... procesar hotplug ...
    udev_device_unref(hotplug_dev);
}
```

> ***El monitor udev se multiplexa con poll() junto con los fds de input.*** *Esto es clave para un programa que escucha tanto eventos de dispositivos como cambios de hotplug sin hilos.*

4.6 Lo que te llevás de esta parte

- **libudev es la capa correcta sobre `/dev/input/`.** Te da nombres, vendors, tipos y hotplug antes de que abras un solo fd.
- **Las propiedades `ID_INPUT_*` clasifican dispositivos.** No hay adivinar con bits.
- **ioctl como fallback siempre.** Si udev no tiene la propiedad, usá `EVIOCGBIT` para clasificar.
- **Hotplug con `udev_monitor`.** Escuchá conexiones/desconexiones en tiempo real con `poll()`.

Práctica de la Parte 4

Ejercicio 26.4 — Enumerar y clasificar dispositivos con libudev (con fallback ioctl)

Conceptos · `libudev`, `udev_enumerate`, propiedades de dispositivos, `EVIOCGBIT` **como fallback**, bitmask

Escribí un programa que liste **todos los dispositivos de entrada conectados**, usando libudev, mostrando:

- Nombre del dispositivo (`DEVNAME`)
- Tipo clasificado con el método primario y secundario
- Vendor y Product ID (si están disponibles)

Flujo de clasificación por dispositivo (ordenado por prioridad):

1. **Método A — libudev:** Chequeá propiedades udev en este orden:
   - `ID_INPUT_KEYBOARD == "1"` → `[keyboard]`
   - `ID_INPUT_MOUSE == "1"` → `[mouse]`
   - `ID_INPUT_JOYSTICK == "1"` → `[joystick]`
   - `ID_INPUT_TOUCHPAD == "1"` → `[touchpad]`
2. **Método B — ioctl fallback:** Si udev no clasificó, abrí el fd con `DEVNAME` y leé los tipos soportados:
   - Solo `EV_KEY` → `[keyboard]`
   - `EV_KEY + EV_REL` sin `EV_ABS` → `[mouse]`
   - `EV_ABS` presente → `[joystick]`
   - Nada claro → `[other]`

Salida esperada:

```
Dispositivos en /dev/input/:
  event0   [char dev]
  event1   [char dev]
  event2   [char dev]
  mice     [regular file]
  mouse0   [char dev]
  ...

📡 Dispositivos de entrada clasificados:
  [keyboard] AT Translated Set 2 Keyboard       → /dev/input/event3  (udev)
  [mouse]    Logitech G502 HERO                 → /dev/input/event7  (udev)
  [joystick] Xbox Wireless Controller           → /dev/input/event12 (udev)
  [other]    USB Generic HID Device             → /dev/input/event9  (ioctl fallback)
```

Criterios de aceptación:

- ☐ Usás `libudev` como método principal de enumeración y clasificación.
- ☐ Implementás el fallback con `EVIOCGBIT` que se ejecuta solo cuando udev no clasifica.
- ☐ Mostrás entre paréntesis el método usado: `(udev)` o `(ioctl fallback)`.
- ☐ Manejás el caso donde no hay dispositivos conectados (output: "No hay dispositivos de entrada").

PARTE 5 — El protocolo de eventos evdev

5.1 Tipos de eventos en detalle

El kernel input subsystem usa un sistema de tipos y códigos bien definido. Acá están los que vas a usar:

EV_KEY (tipo = 1): Teclas y botones

Código	Significado
`KEY_A` ... `KEY_Z`	Teclas A-Z
`KEY_0` ... `KEY_9`	Dígitos
`KEY_ENTER`, `KEY_ESC`, `KEY_TAB`, `KEY_BACKSPACE`	Teclas de control
`KEY_LEFTSHIFT`, `KEY_RIGHTSHIFT`, `KEY_LEFTCTRL`, `KEY_RIGHTCTRL`	Modificadores
`KEY_LEFTALT`, `KEY_RIGHTALT`, `KEY_LEFTMETA`, `KEY_RIGHTMETA`	Modificadores
`BTN_LEFT`, `BTN_MIDDLE`, `BTN_RIGHT`, `BTN_SIDE`, `BTN_EXTRA`	Botones del mouse
`BTN_TRIGGER`, `BTN_THUMB`, `BTN_TOP`, `BTN_TOP2`	Botones del joystick

EV_REL (tipo = 3): Movimiento relativo

Código	Significado
`REL_X`	Movimiento horizontal del mouse
`REL_Y`	Movimiento vertical del mouse
`REL_WHEEL`	Scroll vertical
`REL_HWHEEL`	Scroll horizontal
`REL_RX`, `REL_RY`, `REL_RZ`	Rotación relativa (joystick giratorio)

EV_ABS (tipo = 3): Posición absoluta

Código	Significado
`ABS_X`, `ABS_Y`	Joystick X/Y (stick izquierdo)
`ABS_Z`, `ABS_RZ`	Trigger izquierdo/derecho (L2/R2 o RT/LT)
`ABS_THROTTLE`, `ABS_RUDDER`	Throttle y rudder (volante)
`ABS_HAT0X`, `ABS_HAT0Y`	D-pad (cruceta direccional)

EV_SYN (tipo = 0): Sincronización

Código	Significado
`SYN_REPORT`	Todos los eventos anteriores forman un frame completo
`SYN_CONFIG`	La configuración del dispositivo cambió
`SYN_MT_REPORT`	Fin de un reporte multitouch (para pantallas táctiles)

5.2 Decodificar un evento paso a paso

Acá tenés un ejemplo mínimo que traduce cada evento en texto legible:

```c
#include <stdio.h>
#include <linux/input.h>
#include <string.h>

void decode_event(struct input_event *ev) {
    switch (ev->type) {
        case EV_KEY: {
            // Traducir códigos de tecla a nombre legible
            const char *key_name = "unknown";

            if (ev->code >= KEY_A && ev->code <= KEY_Z)
                key_name = "A+";  // Se calcula en runtime: 'A' + (ev->code - KEY_A)

            // Para EV_KEY, value indica estado: 0=soltado, 1=presionado, 2=repeat
            const char *state;
            if (ev->value == 0) state = "released";
            else if (ev->value == 1) state = "pressed";
            else state = "repeat";

            printf("EV_KEY  %-8s → %s\n", key_name, state);
            break;
        }

        case EV_REL: {
            // Para EV_REL, value es un delta (cuánto se movió desde el último evento)
            printf("EV_REL  %-8s %+d\n",
                   ev->code == REL_X ? "REL_X" :
                   ev->code == REL_Y ? "REL_Y" :
                   ev->code == REL_WHEEL ? "REL_WHEEL" : "?",
                   ev->value);
            break;
        }

        case EV_ABS: {
            // Para EV_ABS, value es una posición absoluta dentro del rango del
dispositivo
            printf("EV_ABS  %-8s → %d\n",
                   ev->code == ABS_X ? "ABS_X" : "?",
                   ev->value);
            break;
        }

        case EV_SYN: {
            // Frame completado — todos los eventos de este tick están procesados
            printf("EV_SYN  SYN_REPORT → frame completo\n");
```

```
            break;
        }
    }
}
```

> ***Nunca procesás un evento aislado.*** *Siempre mirás el patrón:* `EV_KEY + KEY_A (1)` *seguido de* `EV_SYN` *significa "el usuario presionó A en este frame". El SYN es el delimitador de cada tick.*

5.3 Combinaciones de teclas: detectar modificadores

Una combinación como `Ctrl+C` es simplemente dos `EV_KEY` con `value=1` :

1. `KEY_LEFTCTRL` pressed (value=1)
2. `KEY_C` pressed (value=1)

Para detectar combinaciones, mantenés un bitmask de teclas presionadas:

```c
#define KEY_STATE_WORDS ((KEY_MAX / 32) + 1)
uint32_t key_state[KEY_STATE_WORDS];  // bitmask de teclas actualmente presionadas

void update_key_state(struct input_event *ev) {
    if (ev->type != EV_KEY) return;

    int word = ev->code / 32;
    int bit = ev->code % 32;

    // Solo procesamos press (1) y release (0), ignoramos repeat (2)
    if (ev->value == 1) {
        key_state[word] |= (1U << bit);   // presionó → setear bit
    } else if (ev->value == 0) {
        key_state[word] &= ~(1U << bit);  // soltó → limpiar bit
    }
}

// Verificar Ctrl+C:
int is_ctrl_c(void) {
    return (key_state[KEY_LEFTCTRL / 32] & (1U << (KEY_LEFTCTRL % 32))) &&
           (key_state['C' - KEY_A + ('A' / 32)] & (1U << (('C' - KEY_A) % 32)));
}

// Verificar cualquier combinación:
int is_key_combo(uint16_t key1, uint16_t key2) {
    int w1 = key1 / 32, b1 = key1 % 32;
    int w2 = key2 / 32, b2 = key2 % 32;
    return (key_state[w1] & (1U << b1)) && (key_state[w2] & (1U << b2));
}
```

> ***Ojo con el repeat:*** *los eventos de "repeat" (`value=2`) NO deben agregar ni quitar del bitmask — ya está presionado. Solo procesá `value == 1` para el primer press y `value == 0` para release.*

5.4 Lo que te llevás de esta parte

- **Los tipos son tres: KEY, REL, ABS.** KEY = botones/teclas (0/1/repeat), REL = deltas (movimiento del mouse), ABS = posiciones absolutas (joystick).
- **SYN_REPORT separa frames.** Todo lo que viene antes de un SYN_REPORT pertenece al mismo tick.
- **Las combinaciones se detectan con bitmask.** Mantenés un array de `uint32_t` donde cada bit representa una tecla: presionado o no.

Práctica de la Parte 5

Ejercicio 26.5 — Decodificador universal de eventos evdev

Conceptos · `EV_KEY`, `EV_REL`, `EV_ABS`, bitmask, lectura cruda de `/dev/input/eventN`

Escribí un programa que lea eventos de **todos los dispositivos de input** (usando el ejercicio 26.4 para enumerar y clasificar) y decodifique cada evento en texto legible:

```
[05:12:33.456789] event3 KEY A pressed
[05:12:33.512345] event3 KEY A released
[05:12:34.000001] event7 REL_X +3
[05:12:34.000002] event7 REL_Y -1
[05:12:34.100000] event7 BTN_LEFT pressed
[05:12:34.200000] event12 ABS_X +128
[05:12:34.200001] event12 ABS_Y -64
[05:12:34.300000] event12 ABS_Z +32767
[05:12:34.300001] event12 BTN_TRIGGER pressed
```

Criterios de aceptación:

- [ ] Enumerás todos los `eventN` con libudev (ejercicio 26.4).
- [ ] Decodificás correctamente tipos: KEY, REL, ABS.
- [ ] Traducís códigos a nombres legibles (`KEY_A`, `BTN_LEFT`, `ABS_X`).
- [ ] Para EV_KEY, mostrás "pressed"/"released"/"repeat".
- [ ] Para EV_REL, mostrás el delta con signo `+N` / `-N`.

PARTE 6 — Mouse: movimiento relativo y botones

6.1 El mouse como dispositivo evdev

Un mouse USB típico genera eventos en **varios event nodes** (uno para el mouse base, otro opcional para botones adicionales). Los eventos clave son:

```
Mouse base → /dev/input/eventN:
  EV_KEY + BTN_LEFT/BTN_MIDDLE/BTN_RIGHT    → botones principales
  EV_REL + REL_X                             → movimiento X
  EV_REL + REL_Y                             → movimiento Y
  EV_REL + REL_WHEEL                         → scroll vertical
  EV_REL + REL_HWHEEL                        → scroll horizontal

Mouse gaming → /dev/input/eventM (secundario):
  EV_KEY + BTN_SIDE/BTN_EXTRA                → botones laterales
```

6.2 Leer el mouse crudo

Acá tenés un ejemplo que lee y decodifica eventos de un mouse:

```c
#include <linux/input.h>

int main(void) {
    int fd = open("/dev/input/event7", O_RDONLY | O_NONBLOCK);
    if (fd < 0) return 1;

    struct input_event ev;
    int dx = 0, dy = 0, scroll = 0;

    while (read(fd, &ev, sizeof(ev)) == sizeof(ev)) {
        switch (ev.type) {
            case EV_KEY:
                switch (ev.code) {
                    case BTN_LEFT:
                        printf("Botón izquierdo: %s\n",
                               ev.value ? "pressed" : "released");
                        break;
                    case BTN_RIGHT:
                        printf("Botón derecho: %s\n",
                               ev.value ? "pressed" : "released");
                        break;
                    case BTN_MIDDLE:
                        printf("Botón central: %s\n",
                               ev.value ? "pressed" : "released");
                        break;
                }
                break;

            case EV_REL:
                switch (ev.code) {
                    case REL_X:
                        dx += ev.value;
                        printf("Mouse X delta: %+d (acumulado: %d)\n",
                               ev.value, dx);
                        break;
                    case REL_Y:
                        dy += ev.value;
                        printf("Mouse Y delta: %+d (acumulado: %d)\n",
                               ev.value, dy);
                        break;
                    case REL_WHEEL:
                        scroll += ev.value;
                        printf("Scroll: %+d (acumulado: %d)\n",
                               ev.value, scroll);
                        break;
```

```
                }
                break;

            case EV_SYN:
                // Frame completo — imprimir estado actual
                printf("[cursor dx=%+d dy=%+d | buttons: L?M?R? | scroll=%d]\n",
                       dx, dy, scroll);
                break;
        }
    }

    close(fd);
    return 0;
}
```

6.3 El botón de rueda (scroll) como tecla

Curiosamente, `BTN_SIDE` y `BTN_EXTRA` se comportan como botones normales (`EV_KEY`). Solo el scroll vertical (`REL_WHEEL`) es relativo porque mide **cuántas líneas se movió**, no una posición absoluta.

> ***Nota sobre la dirección del scroll:*** *Los valores de* `REL_WHEEL` *son opuestos a lo que mucha gente espera:*
>
> - `+1` *: scroll hacia abajo (alejado del usuario, "down")*
> - `-1` *: scroll hacia arriba (hacia el usuario, "up")*
>
> *Esto viene del protocolo X11 original y se mantiene por compatibilidad.*

6.4 Lo que te llevás de esta parte

- **El mouse reporta movimiento como deltas REL**, no posiciones absolutas. Tenés que acumularlos.
- **Los botones son eventos KEY con BTN_LEFT, BTN_RIGHT, etc.** Value=1 = presionado, value=0 = soltado.
- **REL_WHEEL tiene dirección opuesta a lo intuitivo** por compatibilidad histórica con X11.

Práctica de la Parte 6

Ejercicio 26.6 — Monitor de mouse en tiempo real

Conceptos · `EV_KEY` + botones, `EV_REL` + movimiento, acumulación de deltas

Leé el dispositivo mouse desde `/dev/input/eventN` y mostrá en cada evento:

- Estado actual del cursor (acumulación de deltas X/Y)
- Estado de cada botón (presionado o no)
- Posición acumulada del scroll

```
Mouse state: dx=+42 dy=-7 | Buttons: [L:down M:up R:down] | Scroll: +3
Mouse state: dx=+15 dy=-2 | Buttons: [L:up M:up R:down]    | Scroll: +3
Mouse state: dx=+15 dy=-2 | Buttons: [L:up M:up R:up]      | Scroll: +4
```

Criterios de aceptación:

- ☐ Detectás automáticamente el dispositivo mouse por sus propiedades udev (ejercicio 26.4).
- ☐ Acumulás deltas de X/Y correctamente (no reseteás entre eventos).
- ☐ Mostrás el estado de botones como bitmap: `[L:down M:up R:down]`.

PARTE 7 — Joystick: ejes absolutos y botones

7.1 El joystick como dispositivo evdev

Un gamepad moderno (Xbox, PlayStation, Logitech) expone múltiples ejes y botones a través del mismo `/dev/input/eventN`:

```
Joystick → /dev/input/eventN:
  EV_ABS + ABS_X       → Eje horizontal izquierdo (stick L)
  EV_ABS + ABS_Y       → Eje vertical izquierdo (stick L)
  EV_ABS + ABS_Z       → Trigger izquierdo (L2 en PS, RT en Xbox)
  EV_ABS + ABS_RZ      → Trigger derecho (R2 en PS, LT en Xbox)
  EV_ABS + ABS_THROTTLE→ Pedal (si hay volantes)
  EV_ABS + ABS_HAT0X,Y → D-pad (cruceta direccional)

Joystick buttons:
  EV_KEY + BTN_SOUTH    → Botón "A" / "B" según región
  EV_KEY + BTN_EAST     → Botón "B" / "A"
  EV_KEY + BTN_NORTH    → Botón "X" / "Y"
  EV_KEY + BTN_WEST     → Botón "Y" / "X"
  EV_KEY + BTN_TL       → L1 (shoulder izquierdo)
  EV_KEY + BTN_TR       → R1 (shoulder derecho)
  EV_KEY + BTN_START    → Botón "Start" / "Options"
  EV_KEY + BTN_SELECT   → Botón "Select" / "Share"
  EV_KEY + BTN_THUMB    → L3 (click del stick izquierdo)
  EV_KEY + BTN_THUMB2   → R3 (click del stick derecho)
```

7.2 Calibrar ejes con `EVIOCGABS`

Cada eje tiene un **rango** y **parámetros de calibración**. Podés consultarlo con `EVIOCGABS`:

```c
struct input_absinfo abs;
ioctl(fd, EVIOCGABS(ABS_X), &abs);

printf("Eje ABS_X:\n");
printf("  min:     %d\n", abs.minimum);      // valor mínimo del eje
printf("  max:     %d\n", abs.maximum);      // valor máximo del eje
printf("  fuzz:    %d\n", abs.fuzz);         // ruido aceptable (ignorar cambios menores)
printf("  flat:    %d\n", abs.flat);          // zona muerta central (deadzone)
printf("  resolution %d\n", abs.resolution);  // unidades por mm (o grado, etc.)
```

> ***No asumas que todos los ejes van de `-32768` a `+32767`.*** *Joysticks baratos, pads de guitarra, volantes — cada uno puede tener rangos distintos. Siempre consultá con `EVIOCGABS`.*

7.2.1 La deadzone: ignorar el drift del stick central

Casi todos los joysticks tienen una **deadzone**: una zona pequeña alrededor del centro donde el eje no debería moverse, pero por imperfecciones del hardware sí reporta valores pequeños (drift). El kernel reporta esta zona en `abs.flat`:

```c
int is_centered(int abs_value, struct input_absinfo *abs) {
    // Si el valor está dentro de la deadzone, lo consideramos centrado
    return abs(abs_value) < abs->flat;
}

// Normalizar un eje al rango [-100, +100] ignorando la deadzone:
int normalize_axis(int raw_value, struct input_absinfo *abs) {
    // Si está en la deadzone, retornar 0
    if (is_centered(raw_value, abs)) return 0;

    // Mapear al rango [-100, +100], manteniendo el signo
    int range = abs->maximum - abs->minimum;
    float normalized = ((float)(raw_value - abs->minimum) / range - 0.5f) * 200.0f;

    // Clamp a [-100, +100]
    if (normalized > 100.0f) normalized = 100.0f;
    if (normalized < -100.0f) normalized = -100.0f;

    return (int)normalized;
}
```

> ***¿Por qué normalizar?*** *Porque un joystick puede reportar `ABS_X` de `-32768` a `+32767`, otro de `0` a `65535`, y otro de `1000` a `64000`. Normalizar al rango `[-100, +100]` te da un valor independiente del dispositivo.*

7.2.2 Mapear botones del joystick a nombres legibles

```
const char* get_key_name(uint16_t code) {
    switch (code) {
        case BTN_SOUTH:   return "SOUTH";   // A/B según región
        case BTN_EAST:    return "EAST";    // B/A
        case BTN_NORTH:   return "NORTH";   // X/Y
        case BTN_WEST:    return "WEST";    // Y/X
        case BTN_TL:      return "TL";      // L1 (shoulder izq)
        case BTN_TR:      return "TR";      // R1 (shoulder der)
        case BTN_START:   return "START";
        case BTN_SELECT:  return "SELECT";
        case BTN_THUMB:   return "THUMB";   // L3 (stick click izq)
        case BTN_THUMB2:  return "THUMB2";  // R3 (stick click der)
        default:          return "UNKNOWN";
    }
}

const char* get_axis_name(uint16_t code) {
    switch (code) {
        case ABS_X:     return "X (L-stick H)";
        case ABS_Y:     return "Y (L-stick V)";
        case ABS_Z:     return "Z (LT trigger)";
        case ABS_RZ:    return "RZ (RT trigger)";
        case ABS_HAT0X: return "HAT0X (D-pad H)";
        case ABS_HAT0Y: return "HAT0Y (D-pad V)";
        default:        return "UNKNOWN";
    }
}
```

7.3 Leer el joystick crudo en acción

Acá tenés un ejemplo completo de lectura de un gamepad:

```c
#include <linux/input.h>

#define MAX_AXES 12

typedef struct {
    int fd;
    char name[256];
    struct input_absinfo abs_info[MAX_AXES];  // parámetros de cada eje
    uint32_t key_state[(KEY_MAX / 32) + 1];   // bitmask de botones
} JoystickContext;

void init_joystick(JoystickContext *ctx, const char *path) {
    ctx->fd = open(path, O_RDONLY | O_NONBLOCK);
    ioctl(ctx->fd, EVIOCGNAME(sizeof(ctx->name)), ctx->name);

    // Obtener parámetros de cada eje soportado
    for (int i = ABS_X; i <= MAX_AXES; i++) {
        ioctl(ctx->fd, EVIOCGABS(i), &ctx->abs_info[i - ABS_X]);
    }
}

void poll_joystick(JoystickContext *ctx) {
    struct input_event ev;

    while (read(ctx->fd, &ev, sizeof(ev)) == sizeof(ev)) {
        switch (ev.type) {
            case EV_KEY: {
                int word = ev.code / 32;
                int bit = ev.code % 32;
                if (ev.value == 1) {
                    ctx->key_state[word] |= (1U << bit);    // press
                } else if (ev.value == 0) {
                    ctx->key_state[word] &= ~(1U << bit);   // release
                }
                printf("Joystick %-20s %-8s %s\n",
                       ctx->name, get_key_name(ev.code),
                       ev.value == 1 ? "pressed" : "released");
                break;
            }

            case EV_ABS: {
                int norm = normalize_axis(ev.value, &ctx->abs_info[ev.code - ABS_X]);
                printf("Joystick %-20s %-15s raw=%+6d norm=%+4d\n",
                       ctx->name, get_axis_name(ev.code), ev.value, norm);
                break;
```

```
            }

            case EV_SYN:
                // Frame completo
                break;
        }
    }
}
```

7.4 Lo que te llevás de esta parte

- **El joystick usa ABS para ejes y KEY para botones.** Los ejes reportan posiciones absolutas, no deltas.
- **Siempre consultá `EVIOCGABS`.** El rango puede variar entre dispositivos — nunca asumas `-32768` a `+32767`.
- **La deadzone es real y medible.** Usá `abs.flat` para ignorar el drift del stick centrado.

Práctica de la Parte 7

Ejercicio 26.7 — Monitor de joystick en tiempo real

Conceptos · `EV_ABS`, `EV_KEY`, `EVIOCGABS`, deadzone, normalización

Leé el dispositivo joystick desde `/dev/input/eventN` y mostrá para cada evento:

- Estado de cada eje (valor raw + valor normalizado a `[-100, +100]`)
- Estado de cada botón (presionado o soltado)
- La deadzone calculada con `EVIOCGABS`

```
Joystick: Xbox Wireless Controller
  ABS_X raw=+32800 norm=  +1  flat=12   (centrado, drift ignorado)
  ABS_Y raw=-64    norm=  -1  flat=12   (cerca del centro)
  ABS_Z raw=+65535 norm=+100 flat=0    (trigger a fondo)
  BTN_SOUTH pressed     → Botón A presionado
  BTN_TR released       → R1 soltado
```

Criterios de aceptación:

- [ ] Usás `EVIOCGABS` para obtener el rango y deadzone de cada eje.
- [ ] Normalizás valores al rango `[-100, +100]` respetando la deadzone.
- [ ] Mostrás tanto valor raw como normalizado.
- [ ] Detectás automáticamente el dispositivo joystick por sus propiedades udev (ejercicio 26.4).

PARTE 8 — Ejercicio jefe: capturador universal de input con libinput

Llegó el momento de juntar **todo** lo que aprendiste en un solo programa: enumeración de dispositivos con libudev, lectura cruda de eventos evdev, clasificación keyboard/mouse/joystick, y ahora la capa final: **libinput**.

8.1 ¿Por qué libinput?

Llegados a este punto ya podés leer eventos crudos de cualquier dispositivo. Pero leer `EV_KEY`, `EV_REL` y `EV_ABS` manualmente tiene problemas:

- **No hace calibración automática:** la deadzone del joystick, el sensitivity del mouse... todo lo hacés vos.
- **No normaliza entre dispositivos:** un Xbox Controller y un DualSense reportan ejes distintos.
- **No maneja gestos:** scroll, pinch-to-zoom, tap-to-click... todo hay que calcularlo a mano.
- **No abstrae el backend:** si querés que tu app funcione en X11, Wayland o TTY, tenés que cambiar código.

libinput resuelve todo eso. Es la biblioteca de freedesktop.org para manejo de input devices:

> ***libinput es lo que usa GNOME, KDE y Wayland por debajo.*** *Es el estándar de facto en Linux para manejar teclado, mouse y touchpad. Y además soporta joysticks desde la versión 1.26+.*

Con libinput, no leés `input_event` directamente: creás un `libinput_handle`, le pasás los fds de los dispositivos, y recibís **eventos semánticos**: `LIBINPUT_EVENT_KEYBOARD_KEY`, `LIBINPUT_EVENT_POINTER_MOTION`, `LIBINPUT_EVENT_TOUCH_DOWN`...

8.2 El flujo básico de libinput

```c
#include <libinput.h>

// 1. Crear un contexto libinput (una sola vez)
struct libinput *li = libinput_create_context(&libinput_interface, NULL);

// 2. Crear una "tablet" que agrupa dispositivos del mismo tipo
struct libinput_tablet *tablet;

// 3. Agregar dispositivos al contexto:
// libinput necesita los fds de los dispositivos para poll().
// Usá libudev_get_fd() para obtener el fd desde un device node udev.
libinput_path_add_device(li, "/dev/input/event7");

// 4. Activar el dispositivo:
libinput_device_enable(libinput_device_from_tablet(tablet));

// 5. Bucle principal: leer eventos con libinput_dispatch() + libinput_get_event()
while (1) {
    // poll() espera en los fds que libinput registró
    if (poll(NULL, 0, -1) < 0) break;

    // libinput lee todos los eventos disponibles y los devuelve uno por uno
    struct libinput_event *event = libinput_get_event(li);
    if (!event) break;

    // Procesar el evento según su tipo:
    switch (libinput_event_get_type(event)) {
        case LIBINPUT_EVENT_KEYBOARD_KEY:
            handle_key_event(event);
            break;
        case LIBINPUT_EVENT_POINTER_MOTION:
            handle_mouse_motion(event);
            break;
        case LIBINPUT_EVENT_POINTER_BUTTON:
            handle_mouse_button(event);
            break;
        default:
            break;
    }

    // IMPORTANTE: libinput necesita que confirmes que procesaste el evento
    libinput_event_destroy(event);
}

// 6. Limpieza
```

```
libinput_remove_device(libinput_device_from_tablet(tablet), NULL);
libinput_unref(li);
```

`libinput_dispatch()` + `poll()`: *Este es el patrón fundamental de libinput. No leés eventos directamente del fd — en vez de eso, llamás a `libinput_dispatch()` que lee todos los eventos disponibles y los pone en una cola interna. Después, `libinput_get_event()` saca un evento de la cola uno por uno. Entre medio, usás `poll()` para esperar eficientemente cuando no hay eventos.*

8.3 Teclado con libinput

```c
void handle_key_event(struct libinput_event *event) {
    struct libinput_event_keyboard *k = libinput_event_get_keyboard_event(event);

    // Timestamp en milisegundos (libinput lo normaliza por vos)
    uint32_t time_ms = libinput_event_keyboard_get_time(k);

    // Código de tecla: libinput devuelve el code de evdev directamente
    uint32_t key = libinput_event_keyboard_get_key(k);

    // Estado: down (presionado) o up (soltado)
    enum libinput_config_keyboard_repeating_state repeat;
    if (libinput_event_keyboard_get_key_state(k) == LIBINPUT_KEY_STATE_PRESSED) {
        printf("[%u] KEY 0x%04x DOWN\n", time_ms, key);
    } else {
        printf("[%u] KEY 0x%04x UP\n", time_ms, key);
    }
}
```

libinput devuelve los codes de tecla directamente como evdev codes. *No necesitás traducción: `KEY_A`, `KEY_ENTER`, `KEY_LEFTCTRL` son los mismos valores que ya conocés.*

8.4 Mouse con libinput

```c
void handle_mouse_motion(struct libinput_event *event) {
    struct libinput_event_pointer *p = libinput_event_get_pointer_event(event);

    // Delta de movimiento en mm (libinput normaliza la sensibilidad del device)
    double dx_mm = libinput_event_pointer_get_dx(p);
    double dy_mm = libinput_event_pointer_get_dy(p);

    // ¿Hubo un cambio real de posición?
    if (libinput_event_pointer_has_axis(p, LIBINPUT_POINTER_AXIS_SCROLL_VERTICAL)) {
        double scroll_v = libinput_event_pointer_get_scroll_value_virt(p);
        printf("Scroll vertical: %.2f\n", scroll_v);
    }

    // Convertir mm a unidades lógicas (libinput lo hace automáticamente con dx/dy)
    printf("Motion: %.2f x %.2f mm\n", dx_mm, dy_mm);
}

void handle_mouse_button(struct libinput_event *event) {
    struct libinput_event_pointer *p = libinput_event_get_pointer_event(event);

    // Botón presionado o soltado
    enum libinput_btn btn = libinput_event_pointer_get_button(p);
    enum libinput_button_state state = libinput_event_pointer_get_button_state(p);

    // btn es BTN_LEFT, BTN_RIGHT, etc. — los mismos codes de evdev
    printf("Button %s: %s\n",
           btn == BTN_LEFT ? "LEFT" : btn == BTN_RIGHT ? "RIGHT" : "OTHER",
           state == LIBINPUT_BUTTON_STATE_PRESSED ? "pressed" : "released");
}
```

libinput normaliza la sensibilidad del mouse. *Diferentes mice reportan deltas en unidades distintas (píxeles por movimiento, counts de rueda, etc.). libinput convierte todo a* ***mm*** *y te devuelve* `dx` / `dy` *en esas unidades. La conversión de sensitivity es interna.*

8.5 Joystick con libinput

Desde la versión 1.26+, libinput soporta joysticks/gamspads:

```c
void handle_joystick_event(struct libinput_event *event) {
    struct libinput_event_gesture *g;

    // libinput maneja la deadzone y normalización por vos
    double value = libinput_event_gesture_get_distance(g);  // [0.0, 1.0] normalizado

    // Estado del botón: DOWN o UP
    uint32_t button = libinput_event_gesture_get_button(g);
    enum libinput_button_state state = libinput_event_gesture_get_button_state(g);
}
```

> ***No todos los devices soportados por evdev son soportados por libinput.*** *libinput filtra los dispositivos que conoce. Si un dispositivo genérico no es reconocido, simplemente no aparecerá.*

8.6 Estructura del ejercicio jefe

Escribí una **librería de input en C** que unifique la captura de teclado, mouse y joystick con una API simple basada en callbacks:

```c
// Declaración de callbacks (el usuario los implementa)
typedef void (*key_event_fn)(uint32_t time_ms, uint16_t key_code, int pressed);
typedef void (*mouse_button_fn)(uint32_t time_ms, uint16_t button, int pressed);
typedef void (*mouse_motion_fn)(uint32_t time_ms, double dx_mm, double dy_mm);
typedef void (*joystick_axis_fn)(uint32_t time_ms, int axis_id, float value);
typedef void (*joystick_button_fn)(uint32_t time_ms, uint16_t button, int pressed);

// Estructura principal de la librería
typedef struct {
    struct libinput *li;
    key_event_fn on_key_down;
    key_event_fn on_key_up;
    mouse_button_fn on_mouse_button;
    mouse_motion_fn on_mouse_motion;
    joystick_axis_fn on_joystick_axis;
    joystick_button_fn on_joystick_button;
    int running;  // flag para el loop principal
} input_ctx_t;

// API pública de la librería:
int input_init(input_ctx_t *ctx);                                    // Inicializar
void input_set_callbacks(input_ctx_t *ctx, key_event_fn kdown, ...); // Configurar
callbacks
void input_poll(input_ctx_t *ctx);                                   // Bucle principal
(bloqueante)
void input_shutdown(input_ctx_t *ctx);                               // Limpieza
```

Requisitos obligatorios:

1. **Inicialización automática:** `input_init()` debe encontrar todos los dispositivos de entrada conectados (teclado, mouse, joystick) usando libudev y agregarlos al contexto libinput. Si un dispositivo no es soportado por libinput, debe ignorarlo silenciosamente.
2. **Callbacks configurables:** El usuario define qué hacer cuando llega un evento. La librería NO imprime nada por sí misma — solo llama a los callbacks. Los callbacks reciben:
   - `time_ms` : timestamp en milisegundos (normalizado por libinput)
   - Datos específicos del tipo de evento
3. **Bucle principal con poll():** `input_poll()` usa `poll()` en el fd de libinput para esperar eventos eficientemente. Mientras no haya eventos, el thread duerme. Cuando llega un evento, lo dispatchea y llama al callback correspondiente.
4. **Manejo de hotplug:** Si se conecta o desconecta un dispositivo durante la ejecución:
   - Conexión → agregar automáticamente al contexto libinput
   - Desconexión → remover del contexto y liberar recursos
5. **Sin fugas de memoria:** Todos los objetos libinput deben ser referenciados correctamente. Al llamar a `input_shutdown()` , todo el contexto debe quedar liberado.

Ejemplo de uso (cómo lo usa el usuario):

```c
// El usuario implementa sus callbacks:
void on_key_down(uint32_t time_ms, uint16_t key_code, int pressed) {
    printf("[%u] KEY DOWN 0x%04x\n", time_ms, key_code);
}

void on_mouse_button(uint32_t time_ms, uint16_t button, int pressed) {
    if (pressed)
        printf("[%u] MOUSE BTN %d DOWN\n", time_ms, button);
}

int main(void) {
    input_ctx_t ctx = {0};
    input_set_callbacks(&ctx, on_key_down, NULL, on_mouse_button, NULL, NULL, NULL);

    if (input_init(&ctx) < 0) {
        fprintf(stderr, "Failed to initialize input\n");
        return 1;
    }

    printf("Capturing input... Press Ctrl+C to exit.\n");
    input_poll(&ctx);  // Bucle principal — bloqueante

    input_shutdown(&ctx);
    return 0;
}
```

Criterios de aceptación:

Funcionalidad básica (obligatorio)

- ☐ `input_init()` encuentra y registra automáticamente todos los dispositivos de input.
- ☐ Los callbacks se invocan correctamente para cada tipo de evento (key, mouse, joystick).
- ☐ `input_poll()` bloquea eficientemente con `poll()` cuando no hay eventos.
- ☐ `input_shutdown()` limpia todo sin fugas.

Manejo de combinaciones

- ☐ Se detectan combinaciones de teclas: Ctrl+C, Alt+Tab, Meta+E (implementar un detector básico).
- ☐ El callback de key_down y key_up se invocan en el orden correcto del evento evdev.

Robustez

- [ ] Maneja dispositivos que aparecen/desaparecen entre polls (hotplug).
- [ ] Si libinput no soporta un dispositivo, lo ignora sin crash.
- [ ] No hay fugas de memoria verificadas con valgrind o sanitizers.

Código limpio

- [ ] La API es simple y consistente: `init()`, `poll()`, `shutdown()`.
- [ ] Los callbacks son typedefs claros, no funciones crudas con firmas confusas.
- [ ] La librería encapsula libinput — el usuario final nunca ve `struct libinput_*` directamente.

8.7 Compilación del ejercicio jefe

```
# Necesitás: libinput-dev, libudev-dev
gcc input.c exercise_jefe.c -o input_demo \
    $(pkg-config --cflags --libs libinput libudev)
```

> ***Verificación con valgrind:*** *Corré* `valgrind --leak-check=full ./input_demo` *para confirmar que no hay fugas de memoria. Libinput maneja la mayoría, pero si creás estructuras propias (como* `input_ctx_t`*), asegurate de liberar todo en* `input_shutdown()`*.*

8.8 Lo que te llevás de este capítulo

Resumen rápido de lo que aprendiste (y las herramientas que ahora tenés):

- **El input subsystem es uniforme.** Teclado, mouse, joystick — todo viaja como `input_event` por `/dev/input/eventN`. Un solo formato para todos los dispositivos.
- **Evdev es la capa más baja:** leés `struct input_event`, decodificás type/code/value a mano. Potente pero detallista. Ideal cuando necesitás control total.
- **libudev es la capa de enumeración:** te dice qué dispositivo es cuál, vendor/product, hotplug. Siempre usala antes de abrir un fd.
- **ioctl completa el picture:** `EVIOCGBIT` descubre qué hace un device, `EVIOCGNAME` su nombre, `EVIOCGABS` los parámetros de ejes. Es la red de seguridad cuando udev no alcanza.
- **libinput es la capa de abstracción más alta:** normaliza sensibilidad, deadzones, gestures. Usala para aplicaciones reales; usá evdev directo para herramientas de bajo nivel.

> ***Felicidades.*** *Acabás de escribir un sistema completo de captura de input en Linux — desde los bytes crudos del kernel hasta una API de alto nivel con callbacks. Eso significa que entendés más del subsistema de input de Linux que el 95% de los desarrolladores de videojuegos y apps, porque la mayoría usa SDL o Raylib sin saber qué hay debajo.*

¿Qué sigue?

Este capítulo cierra el recorrido por input en Linux desde cero absoluto hasta libinput. Los próximos capítulos podrían explorar:

- **Input hotplug con daemon:** Un servicio que arranca con el sistema y mantiene lista una tabla de dispositivos activos para reutilizarlos entre sesiones.
- **Event injection:** Escribir eventos de input simulados con `libevdev` o `uinput`.
- **Evdev raw mode:** Desactivar el proceso automático de repeat del kernel y controlar los delays manualmente.
- **Joystick calibration wizard:** Una herramienta interactiva para detectar deadzones y rangos exactos de cada eje.

¡Tu elección! El capítulo sobre input en Linux está completo — las ideas quedan a partir de ahí.