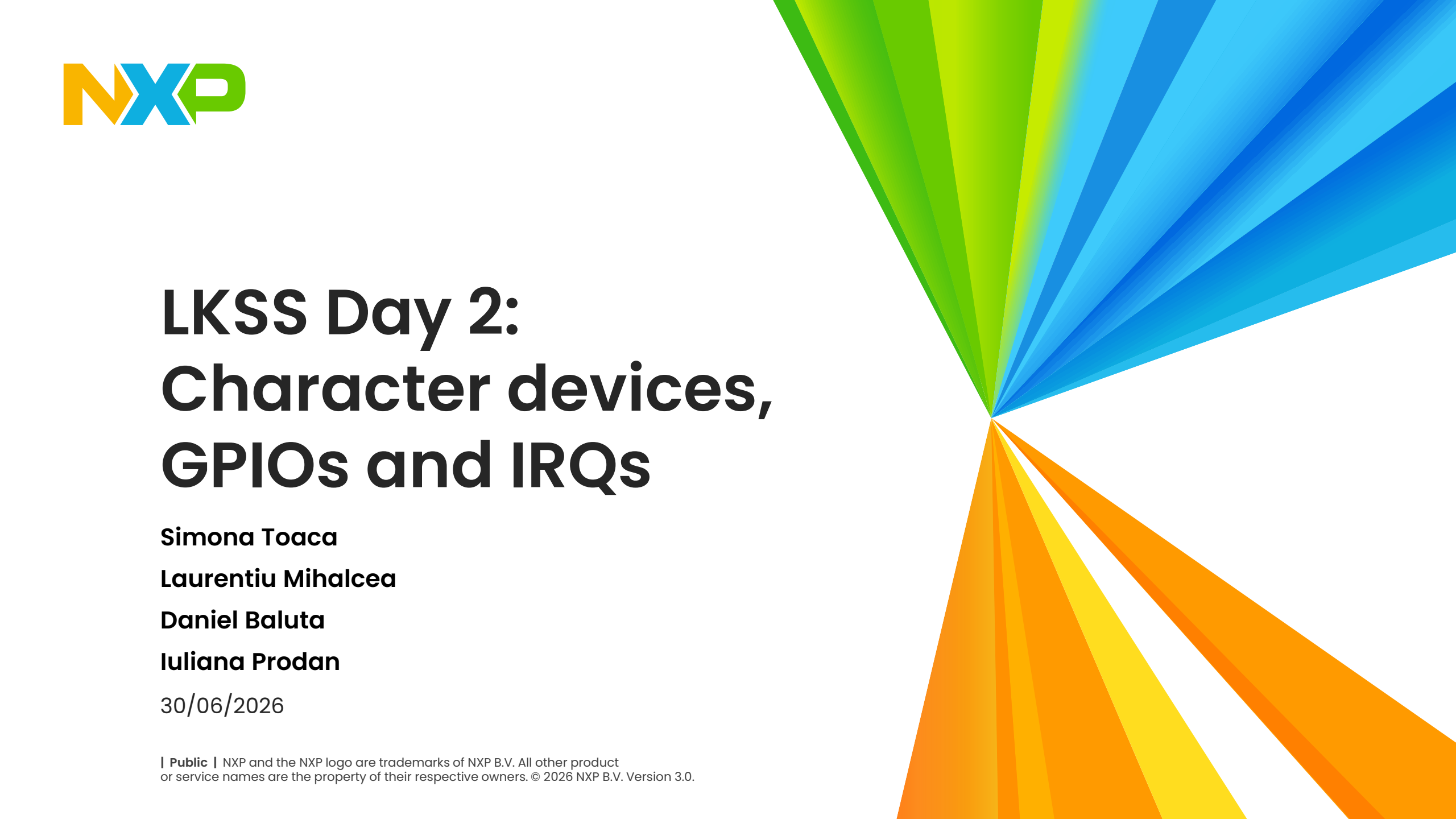


LKSS Day 2: Character devices, GPIOs and IRQs

Simona Toaca

Laurentiu Mihalcea

Daniel Baluta

Iuliana Prodan

30/06/2026

Today’s topics

- **Character** and **Miscellaneous** Devices
- **List** API in Linux
- **Files** in the Linux Kernel
- Using **GPIOs** to interact with LEDs and buttons
- Using **interrupts** to avoid polling
- Correlating the schematic with the Device Tree
- Button **debouncing**

Notes and sources

01

Character and Miscellaneous devices

Device files - /dev

- Interface in **user space** for accessing **device driver** functionalities
- Interaction through **file operations** (open, read, write, ioctl, close, etc.) -> **system calls**
- **2** types: **character** and **block**
- Each device -> identified by **major** and **minor**

```
$ file -L /dev/tty0
/dev/tty0: character special (4/0)
$ file -L /dev/tty1
/dev/tty1: character special (4/1)
$ file -L /dev/cdrom
/dev/cdrom: block special (11/0)
```

Notes and sources

File operations

- For **character** devices -> file operations are mapped directly to **system calls**
- But you may notice…the **signature** is different

```
struct file_operations {
        struct module *owner;
        fop_flags_t fop_flags;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        […]
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *, fl_owner_t id);
        […]
};
```

Notes and sources

Character devices

- Expose an API for **serial** communication -> **character device file**
- **Register** -> **file ops** are attached to the **character device**
- After registering -> no file in /dev **yet**

Registering API:

int **register_chrdev**(unsigned int **major,** const char *name, const struct file_operations ***fops**);

void **unregister_chrdev**(unsigned int **major**, const char *name);

or

int **register_chrdev_region**(**dev_t first**, unsigned int count, char *name);

void **unregister_chrdev_region**(**dev_t first**, unsigned int count);

paired with

void **cdev_init**(struct cdev *cdev, struct file_operations ***fops**);

int **cdev_add**(struct cdev *dev, **dev_t num**, unsigned int count);

void **cdev_del**(struct cdev *dev);

Character devices

- To create a file in user space:

  - **mknod** – mknod /dev/<name> c <**major**> <**minor**>

  or

  - **class_create** + **device_create:**

  struct class *class_create(const char *name);

  void **class_destroy**(const **struct class *cls**);

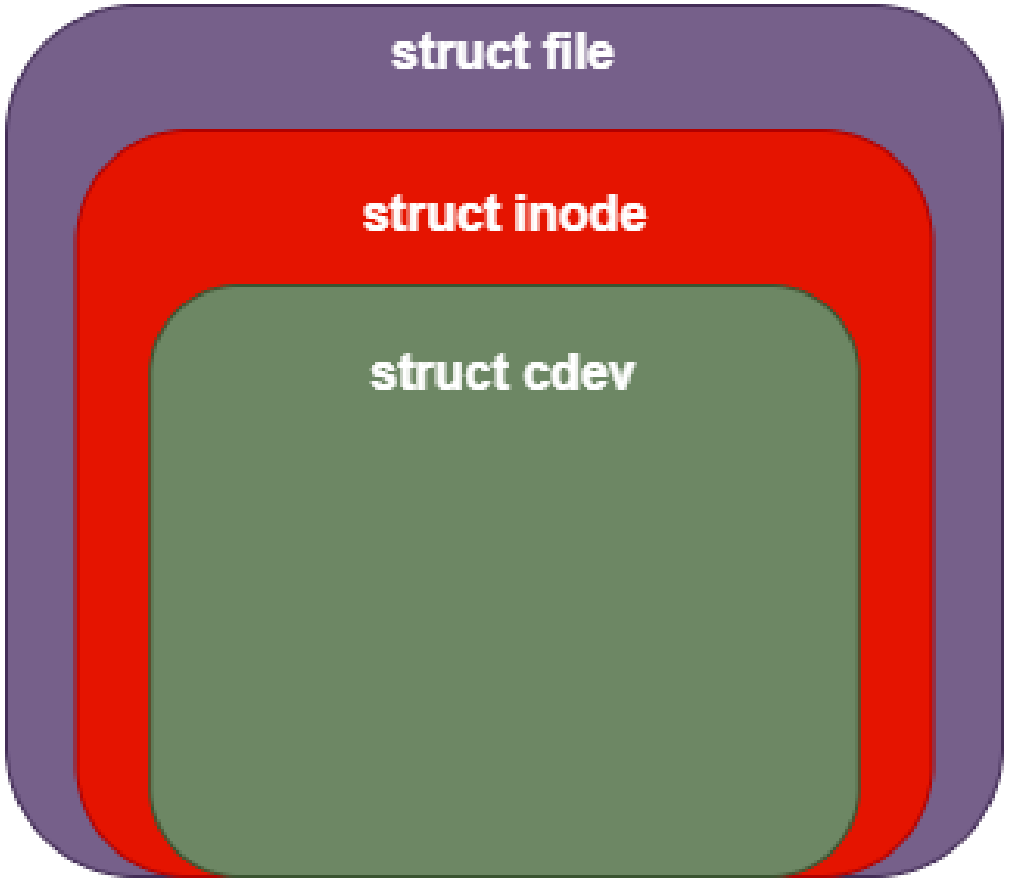
  struct device ***device_create**(const **struct class *cls**, struct device *parent, **dev_t devt**, void *drvdata, const char *fmt, ...);

  void **device_destroy**(const **struct class *cls**, **dev_t devt**);

Notes and sources

File representation in the kernel

- struct **inode** -> can contain a **character device**
- struct **file** -> created upon `**open**` syscall, points to the **inode**
- Possible: **multiple files** pointing to the **same inode**
- file operations -> available on all levels



Notes and sources

Miscellaneous Devices

- **Subset** of character devices
- **Limited** functionality, **simpler** API
- Use of ``struct **miscdevice**``
- major = **10**

API:

```
static struct miscdevice misc_dev {
        .minor = MISC_DYNAMIC_MINOR,
        .name="misc0",
        .fops = &fops
};
int misc_register(struct miscdevice *misc);
void misc_deregister(struct miscdevice *misc);
```

02

API for user space data transfer

Why do we need such an API?

- Dereferencing pointers from user space in kernel space is a **bad idea**.
- Different memory mapping
- **Solution**: dedicated API

Notes and sources

copy_to_user/copy_from_user

- Used for moving data:
  - From a buffer in kernel space **to** a buffer in user space
  - **From** a buffer in user space to a buffer in kernel space

- **API**:

unsigned long **copy_to_user**(void __user *to, const void *from, unsigned long n);

unsigned long **copy_from_user**(void *to, const void __user *from, unsigned long n);

Notes and sources

put_user/get_user

- Macros, not methods
- Used for accessing specific addresses in user space
- Usually implemented in assembly
- **put** value <val> at <address>
- **get** value from <address> and store it in <val>

- **API**:

put_user(type val, type *address);

get_user(type val, type *address);

Notes and sources

03

Lists in the Linux kernel

Lists structure

- **circular**, **doubly-linked** lists
- node = previous node + next node pointers -> **struct list_head**

```
struct list_head {
        struct list_head *next, *prev;
};
```

- Where is the data? -> user declares own node containing the data and a **struct list_head** -> user manages the memory

```
struct my_node {
        uint32_t my_data;
        struct list_head list; -> this is what gets added to the linked list
};
```

List API

LIST_HEAD(struct list_head list) -> static declaration
INIT_LIST_HEAD(struct list_head *list) -> dynamic allocation

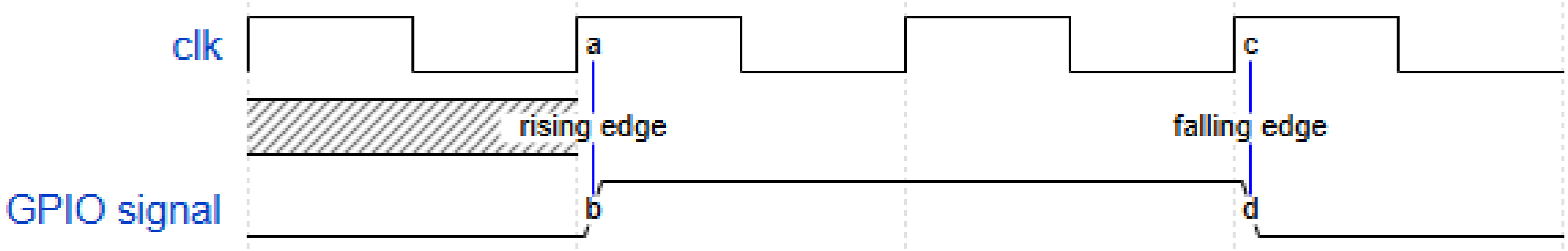
list_add(struct list_head *new, struct list_head *head) -> adds to **next**
list_add_tail(struct list_head *new, struct list_head *head) -> adds to **prev**
list_del(struct list_head *entry)
list_entry(ptr, type, member) -> casts the entry in `ptr` to `type`
list_for_each(pos, head) -> iterate
list_for_each_safe(pos, n, head) -> iterate, safe for node deletion

04

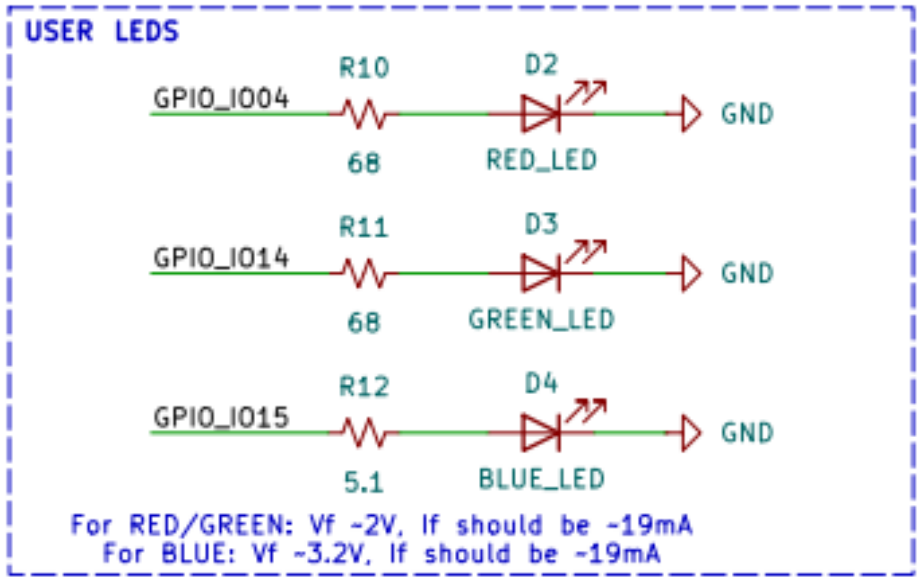
GPIO: General Purpose Input-Output

What are GPIOs?

- General Purpose Input/Output
- **Digital** pins -> 0/1 on input/output
- 0 -> 1 **rising edge**
- 1 -> 0 **falling edge**



GPIOs in the Device Tree



- GPIOs can be described as an array of : <gpio_controller> <gpio_line> <consumer flags>
- Our case: directly correlated to the **schematic**
- Details in the [dtschema]

```
lkss-bus {
        compatible = "simple-bus";

        button-led {
                compatible = "lkss,lab2";
                button-gpios = <&gpio2 22 GPIO_ACTIVE_HIGH>,    // SW4: GPIO2.IO[22]
                               <&gpio2 18 GPIO_ACTIVE_HIGH>;    // SW2: GPIO2.IO[18]
                led-gpios = <&gpio2 4 GPIO_ACTIVE_HIGH>,        // Red led: GPIO_IO[4]
                            <&gpio2 14 GPIO_ACTIVE_HIGH>,       // Green led: GPIO_IO[14]
                            <&gpio2 15 GPIO_ACTIVE_HIGH>;       // Blue led:GPIO_IO[15]
                status = "okay";
        };
};
```

GPIOs in the kernel – gpiod_ API

- To work with GPIOs, you need to get their descriptors: **struct gpio_desc**
- Using the descriptor -> get/set value
- enum **gpiod_flags**: GPIO_IN, GPIO_OUT_LOW, etc.

API:

struct gpio_desc ***gpiod_get_index**(struct device *dev, const char *con_id, unsigned int idx, enum gpiod_flags flags);

struct gpio_descs ***gpiod_get_array**(struct device *dev, const char *con_id, enum gpiod_flags flags);

int **gpiod_get_value**(**struct gpio_desc** *desc);

int **gpiod_set_value**(**struct gpio_desc** *desc, int value);

05

Interrupts

What are interrupts?

- way to **signal events** -> trigger a handler
- avoid busy-waiting
  - While (something) { /* do nothing */ }
- require **hardware support** (usually interrupt controller)
- **interrupt handler** -> **atomic context** (interrupts are masked, no locks allowed, must return ASAP)
- **example**: button press (GPIO) -> trigger an interrupt -> kernel jumps to the routine for button pressing

Interrupt handling in the kernel

- Every interrupt -> has a unique interrupt number
- For GPIOs, request via: **gpiod_to_irq**
- Register an **interrupt handler** for that unique interrupt number: **request_irq**

API:

int gpiod_to_irq(struct gpio_desc *desc);

int **request_irq**(unsigned int **irq**, irq_handler_t handler, unsigned long flags, const char *name, void *dev);

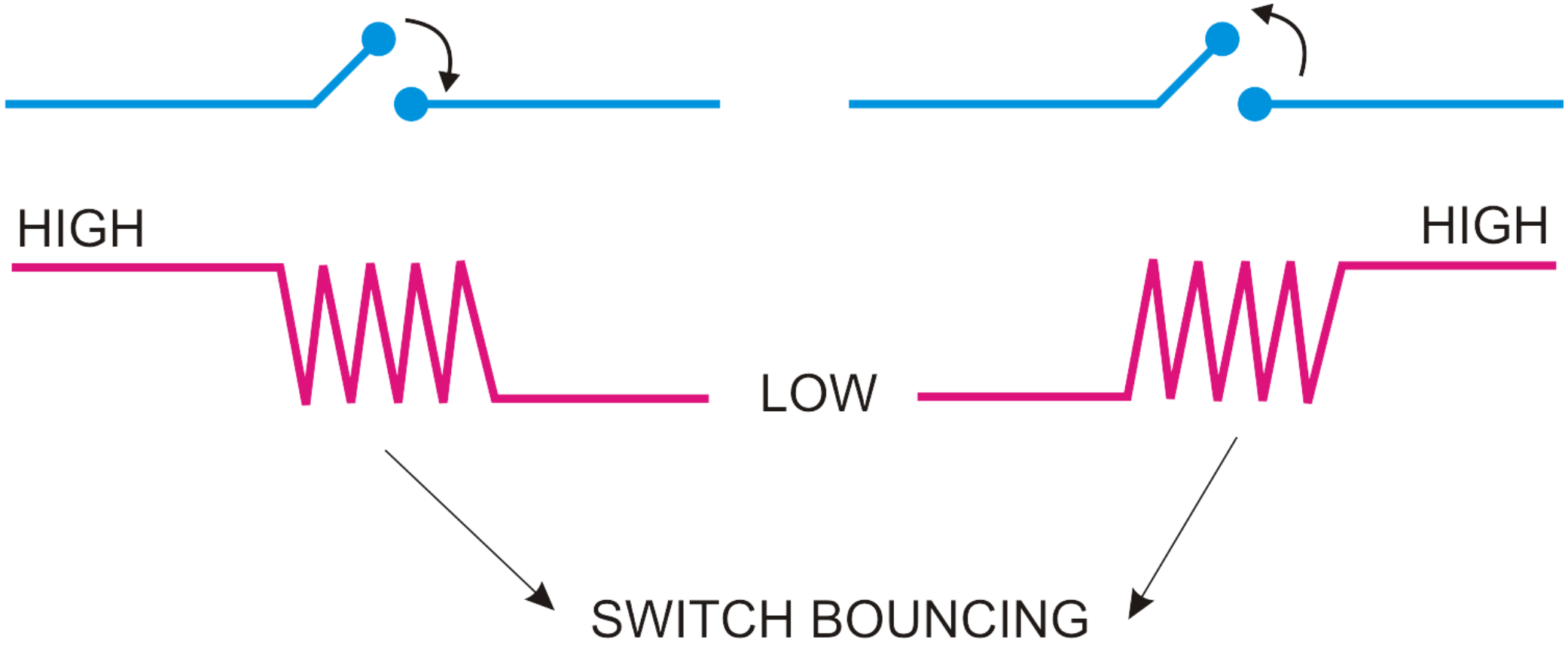
void ***free_irq**(unsigned int irq, void *dev);

06

Debouncing

What Is signal bouncing?

- The rising/falling edge of the signal is not perfect
- The signal ‘bounces’ between 0 and 1 logic levels, triggering multiple interrupts instead of just one



Notes and sources [Arduino Button Debounce]

Button debouncing

- Buttons are susceptible to signal bouncing
- **Debouncing** -> making sure you don't interpret the jitter as multiple signal transitions, by using a **timeout**
- During the timeout interval, all interrupts from the button are ignored
- Timeout ~ 200-300ms
- An example of this:

```
• int irq_handler(...)
• {
        if (current_time < last_press + timeout)
                return IRQ_HANDLED;

        /* Actual irq handling -> when the irq is valid */
}
```

Brighter
Together

nxp.com