Exploring Linux Device Drivers through Raspberry Pi GPIO

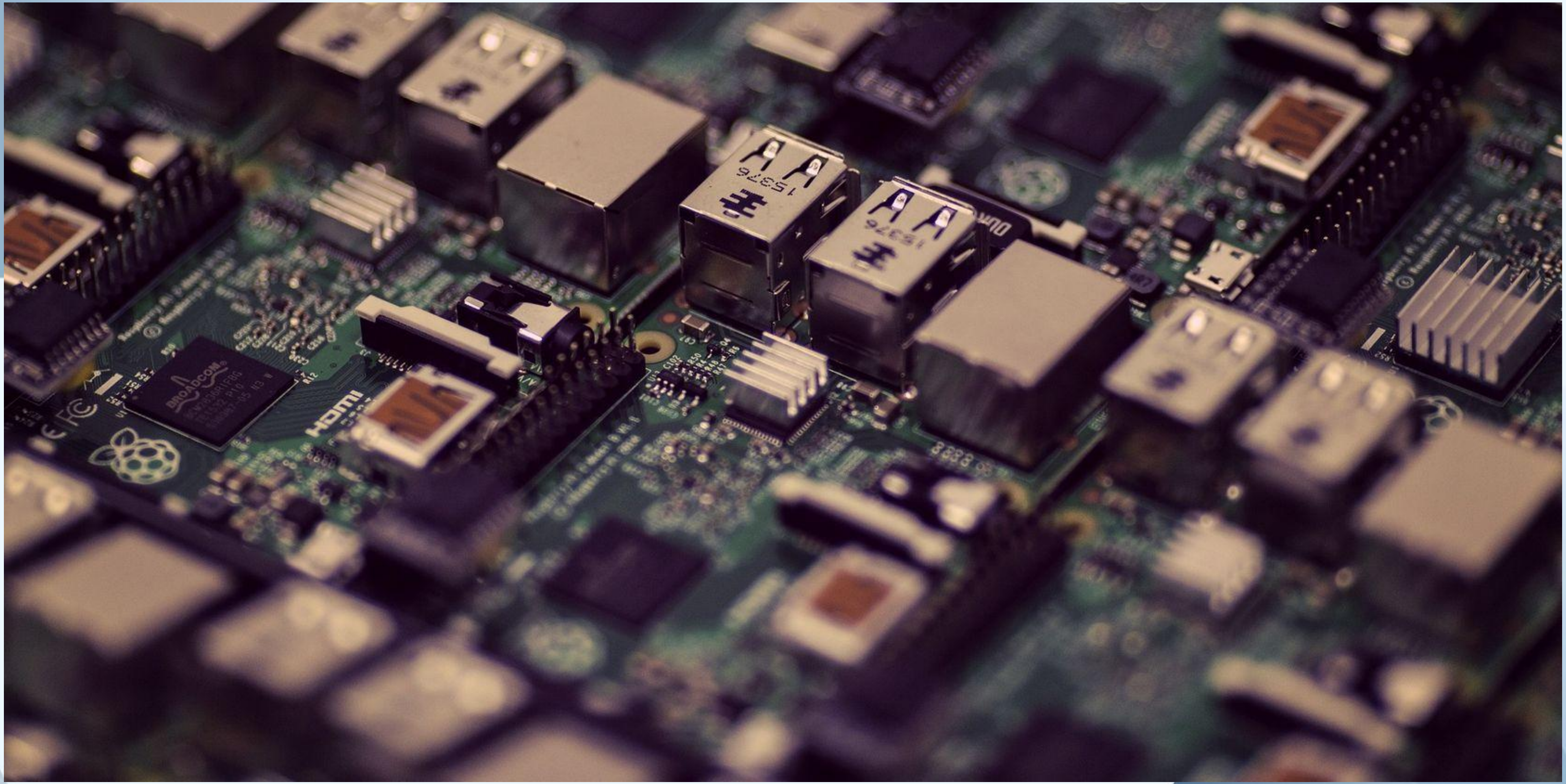
Josue Hernandez: 26 Sep 2023 @ HackerGarage

All the information provided in this presentation is the responsibility of the presenter and not of the company he works for.

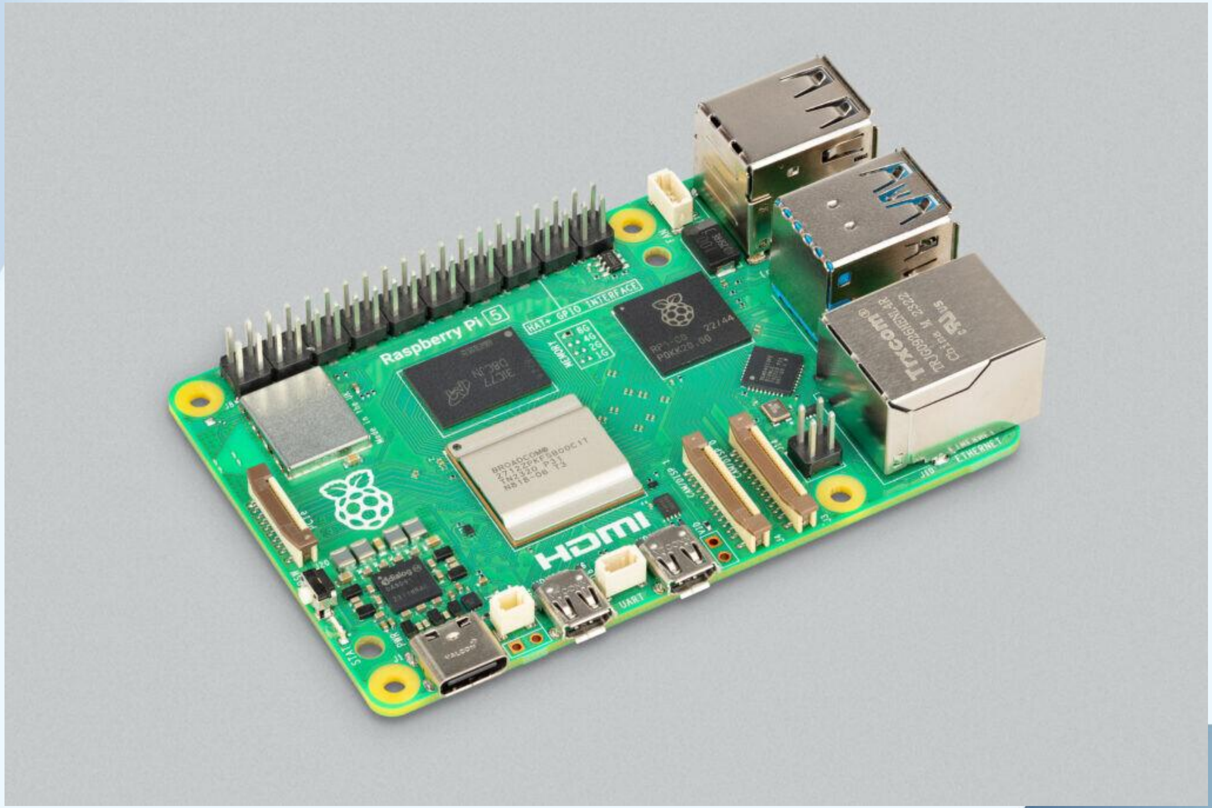
Overview

- Context
  - What Raspberry PI?
  - What is GPIO?
  - What is Linux?
- How can we use GPIO pins on a Linux Machine?
  - GPIO subsystem uapi
  - Custom Kernel Driver
  - LED Class Driver
  - Kernel Bus Infrastructure
  - Hardware description

What is Raspberry PI?



What is Raspberry PI?

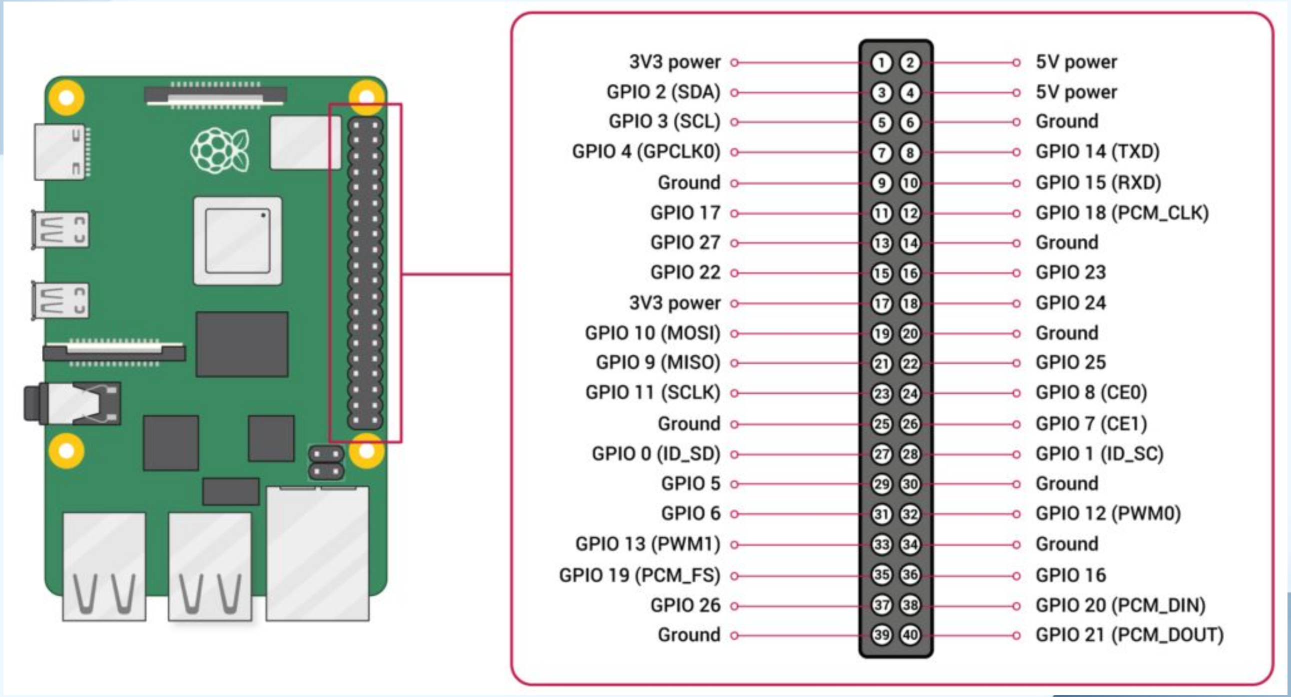


What is Raspberry PI?

- Broadcom BCM2837B0, Cortex-A53 (ARMv8) 64-bit SoC @ 1.4GHz
- 1GB LPDDR2 SDRAM
- 2.4GHz and 5GHz IEEE 802.11.b/g/n/ac wireless LAN, Bluetooth 4.2, BLE
- Gigabit Ethernet over USB 2.0 (maximum throughput 300 Mbps)
- Extended 40-pin GPIO header
- Full-size HDMI®
- 4 USB 2.0 ports
- CSI camera port for connecting a Raspberry Pi camera
- DSI display port for connecting a Raspberry Pi touchscreen display
- 4-pole stereo output and composite video port
- Micro SD port for loading your operating system and storing data
- 5V/2.5A DC power input
- Power-over-Ethernet (PoE) support (requires separate PoE HAT)

What is Raspberry PI?

Raspberry Pi standard 40-pin header



What is GPIO?

- Stands for General Purpose Input Output.
- Signal pin that can be used to perform digital input or output functions
- Commonly grouped in GPIO ports (a group of GPIO pins)
- They can be used in 4 modes
  - GPIO Output Mode
  - GPIO Input Mode
  - Analog Mode
  - Alternate Function mode

What is Linux?

ubuntu

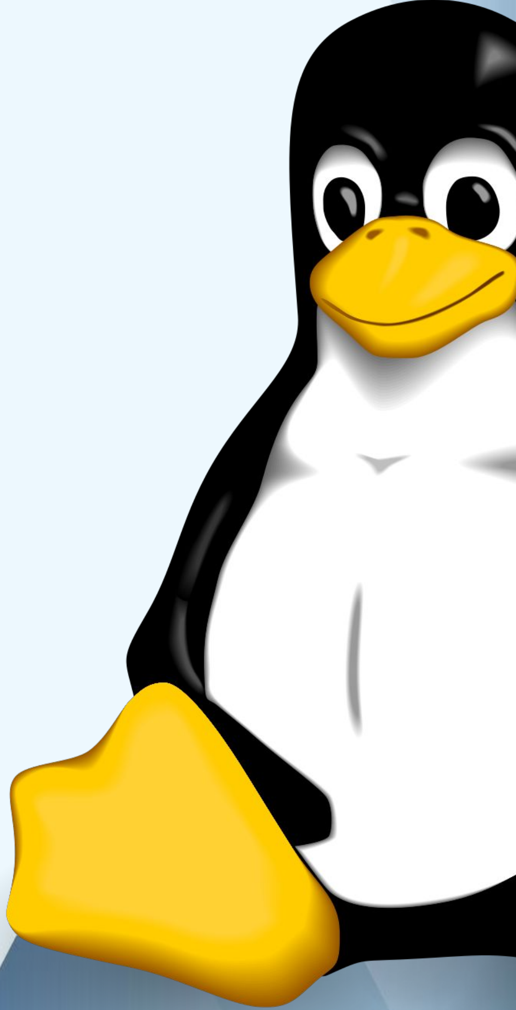
redhat

Clear Linux* Project

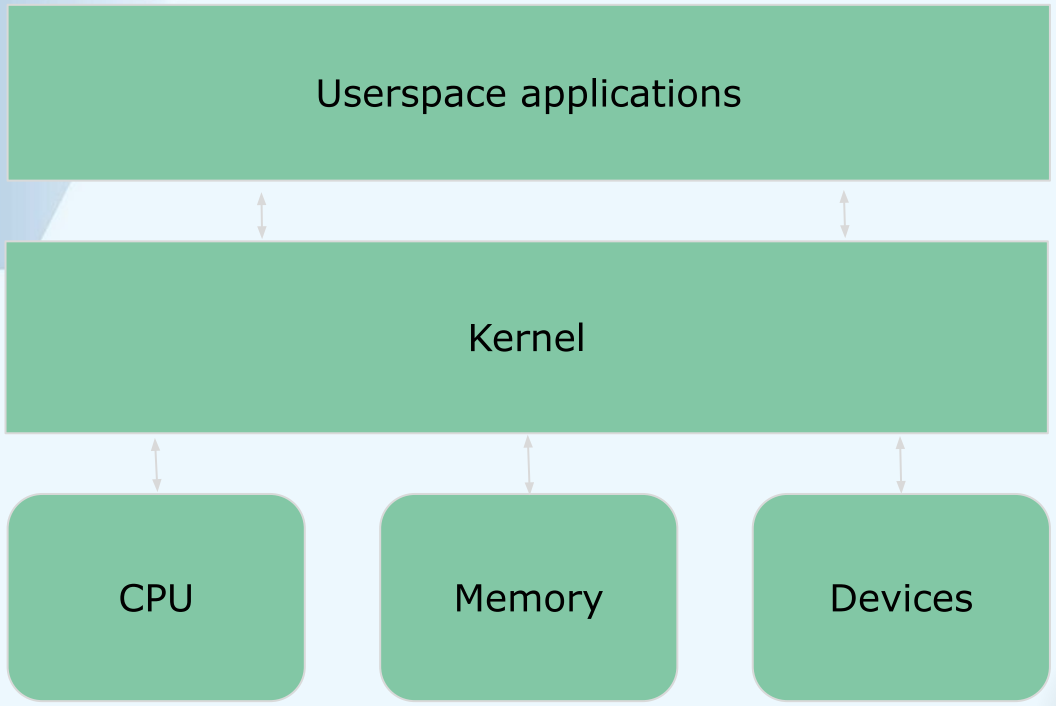
yocto PROJECT

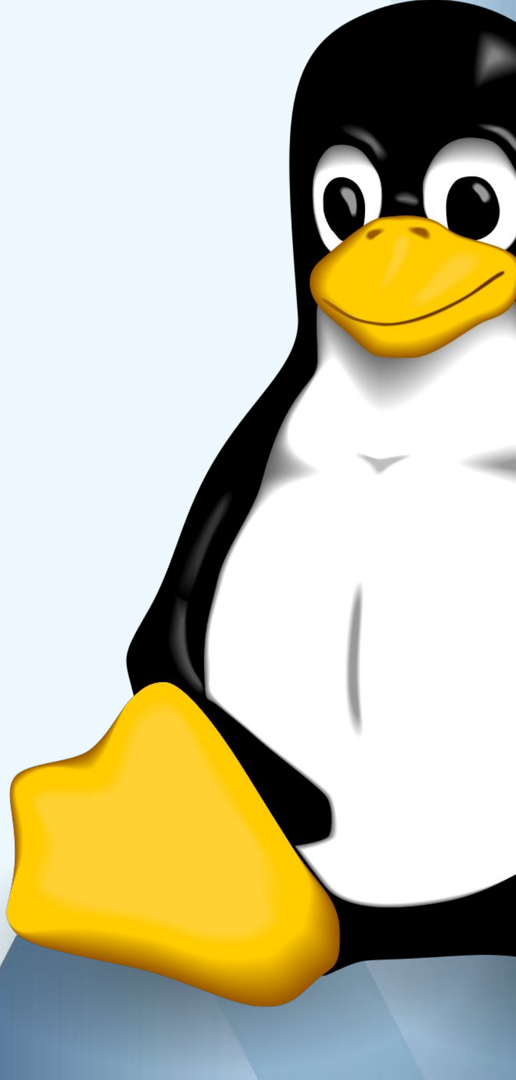
OpenWrt WIRELESS FREEDOM

BuildRoot
Making Embedded Linux Easy

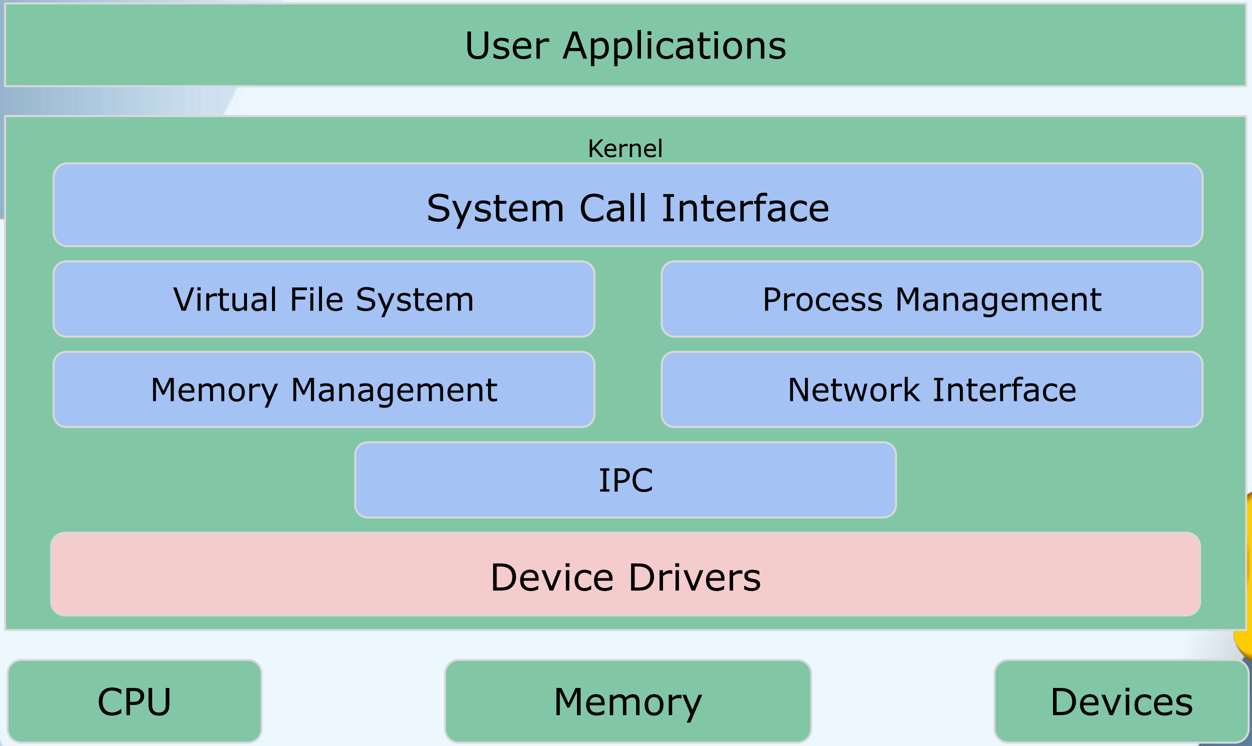


What is Linux?





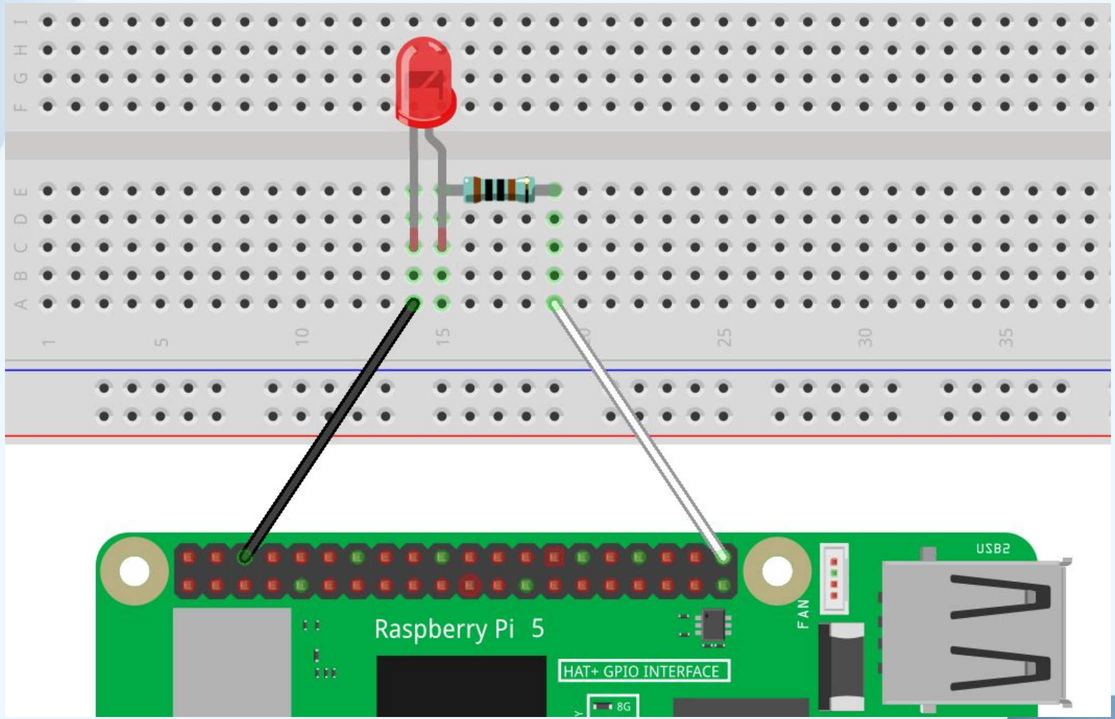
What is Linux?



Overview

- ~~Context~~
  - ~~What Raspberry PI?~~
  - ~~What is GPIO?~~
  - ~~What is Linux?~~
- How can we use GPIO pins on a Linux Machine?
  - GPIO subsystem uapi
  - Custom Kernel Driver
  - LED Class Driver
  - Kernel Bus Infrastructure
  - Hardware description

How can we use GPIO pins on a Linux Machine?



How can we use GPIO pins on a Linux Machine?

Linux Kernel expose GPIO to the user space using the GPIO subsystem.

- Expose a device driver to interact with the whole GPIO port in /dev/gpiochipx (4.8 or newer)
- Provide a list available GPIO pins and its configuration
- Get and Set values to the GPIO pin
- Set even polling for a GPIO pin

GPIO subsystem uapi

GPIOZero is a python library that allow us to work with GPIO.

- Base classes
  - gpiozero.OutputDevice
  - gpiozero.InputDevice
- Regular classes
  - gpiozero.LED
  - gpiozero.Button
  - More others

GPIO subsystem uapi: GPIOZero example

```
#!/usr/bin/env python

from gpiozero import LED


led = LED(21)

led.blink(background=False)
```

```
#!/usr/bin/env python

from gpiozero import OutputDevice

from time import sleep

out = OutputDevice(21)

while True:

    out.on()

    sleep(1)

    out.off()

    sleep(1)
```

Custom Kernel Driver

For writing a Linux Kernel driver we would need the following tools.

- Linux Kernel headers
  - These heder describe kernel source code that can be used in our module
- Cross-compiling tools / a compiler
  - Cross compiler is a compiler that in going to generate binary code to an arch different than the arch of the system where the compiler is running.

Custom Kernel Driver

- Non coding work
  - Define the name of the public interface “myLED”
  - Define the protocol to interact with the LED “1” to turn it on and “0” to turn it off
- Coding work
  - Create the interface to communicate with the LED (character device driver)
    - Create the visible device
    - Create a way to write to the device (write fops)
    - Create a way to read from the device (read fops)
  - Setup the GPIO pin and communicate with it

Custom Kernel Driver

Create the interface to communicate with the LED (character device driver)

```
static int __init myled_driver_init(void)

static void __exit myled_driver_exit(void)


module_init(myled_driver_init);

module_exit(myled_driver_exit);
```

Custom Kernel Driver

Create the interface to communicate with the LED (character device driver)

```
int alloc_chrdev_region(dev_t *dev, unsigned baseminor,
unsigned count, const char *name);
```

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

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

Custom Kernel Driver

Create the interface to communicate with the LED (character device driver)

```
struct class *class_create(const char *name)
```

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

Custom Kernel Driver

Reading and writing to our device driver

```
static inline unsigned long copy_from_user(void *to, const
void __user *from, unsigned long n)

static inline unsigned long copy_to_user(void __user *to,
const void *from, unsigned long n)
```

Setup the GPIO pin and communicate with it

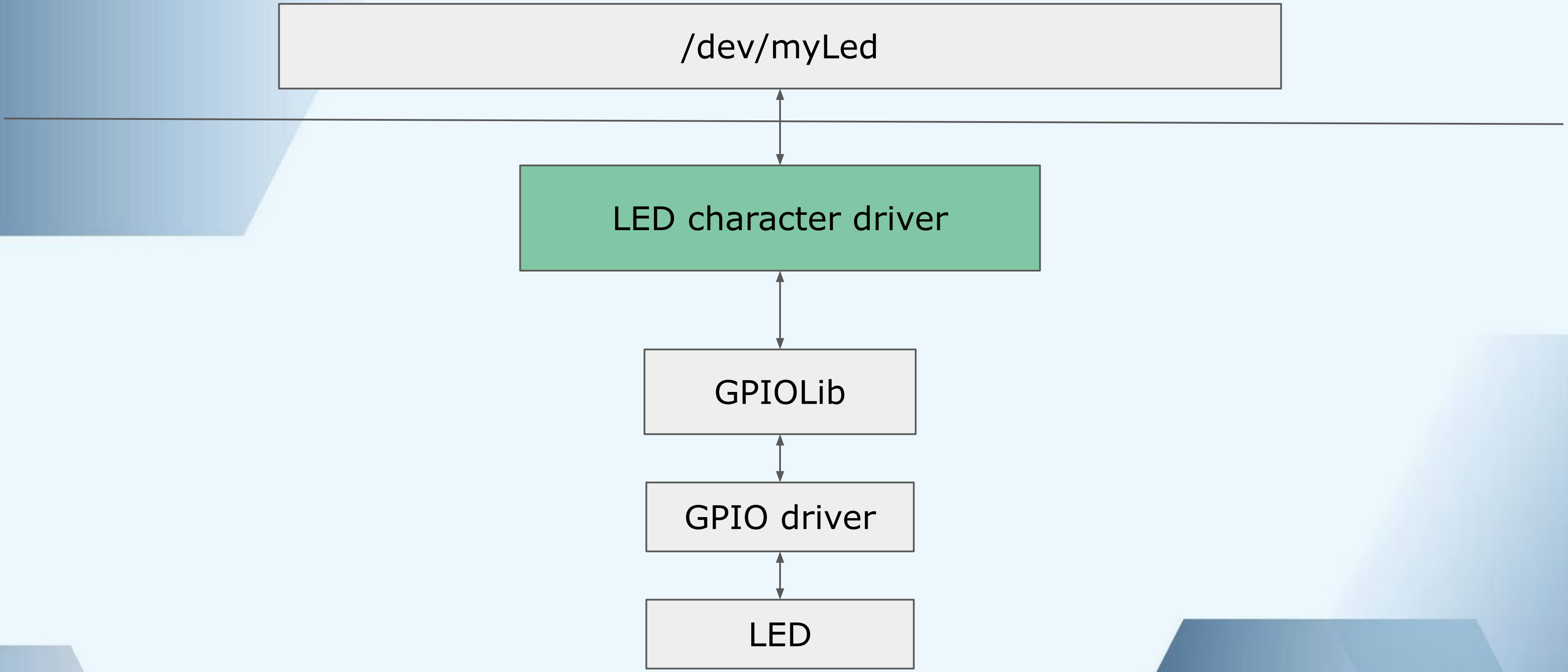
Setup GPIO

```
int gpio_request(unsigned gpio, const char *label)

struct gpio_desc *gpio_to_desc(unsigned gpio)

int gpiod_direction_output(struct gpio_desc *desc, int
value)
```

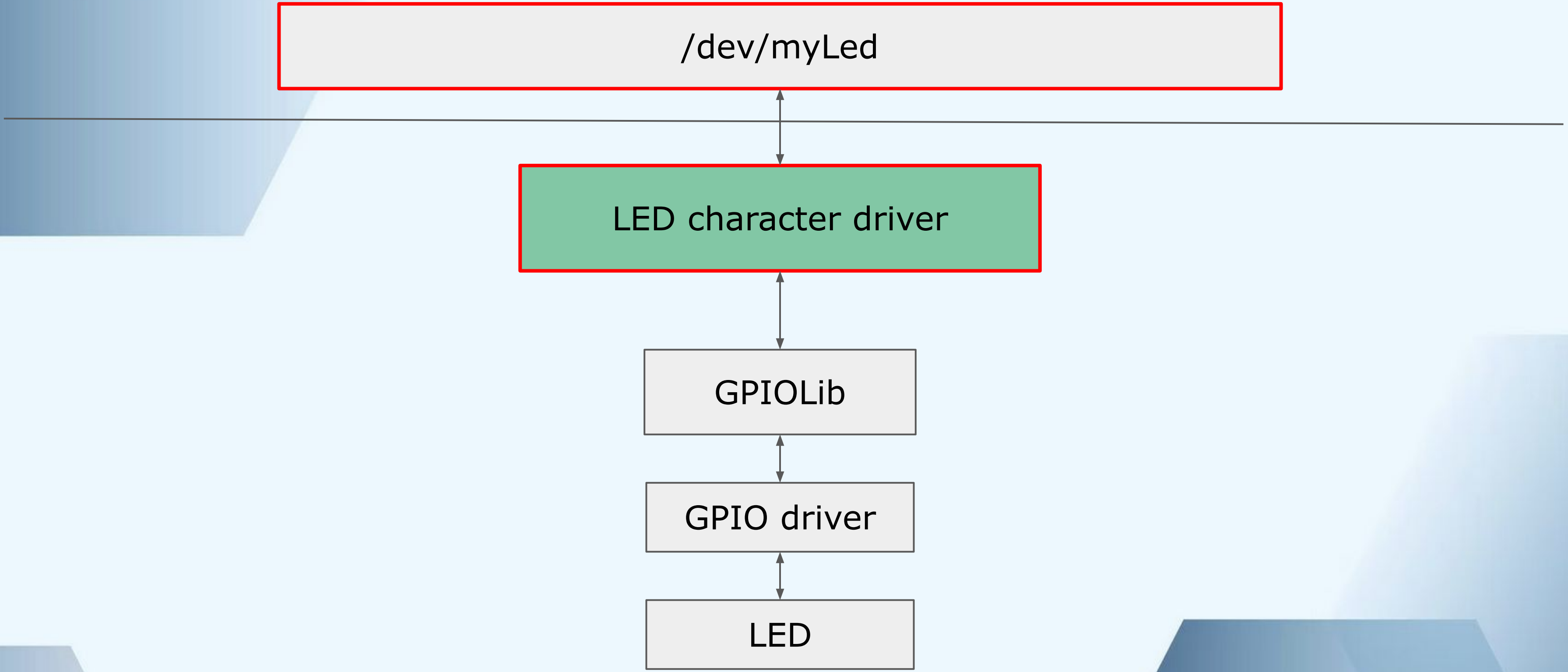
Set and get value of a GPIO pin

```
void gpiod_set_value(struct gpio_desc *desc, int value)

int gpiod_get_value(const struct gpio_desc *desc)
```

Custom Kernel Driver



Custom Kernel Driver



LED Class Driver

- Non coding work
  - Define the name of the public interface “myLED”
  - ~~Define the protocol to interact with the LED “1” to turn it on and “0” to turn it off~~
- Coding work
  - ~~Create the interface to communicate with the LED (character device driver)~~
    - ~~Create the visible device~~
    - ~~Create a way to write to the device (write fops)~~
    - ~~Create a way to read from the device (read fops)~~
  - Setup the GPIO pin and communicate with it

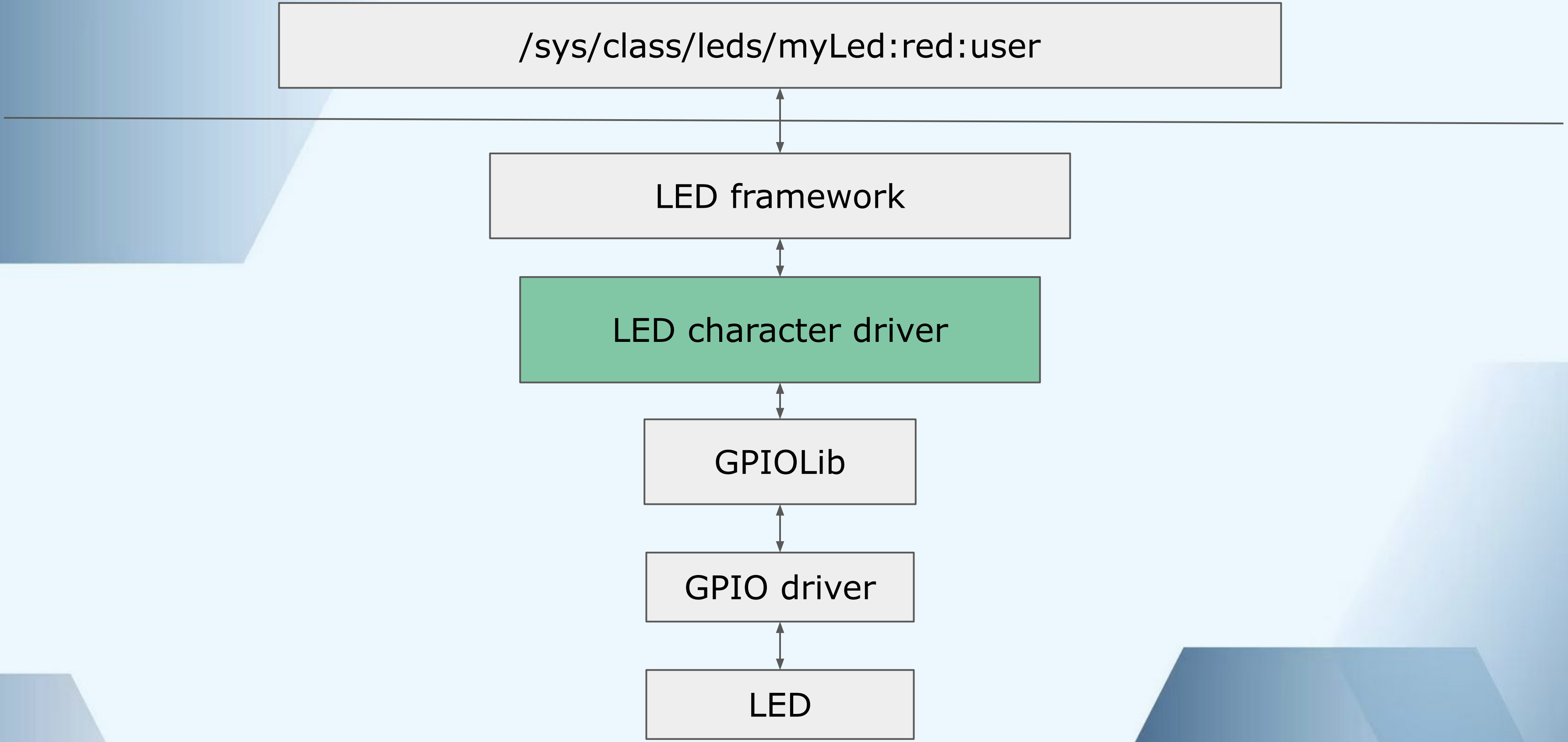
LED Class Driver

```
struct led_classdev led_cdev;

led_cdev.name = "myled:red:user";

led_cdev.brightness_set = myled_on_change_function;

static void myled_on_change_function(struct led_classdev
*led_cdev, enum led_brightness brightness);
```

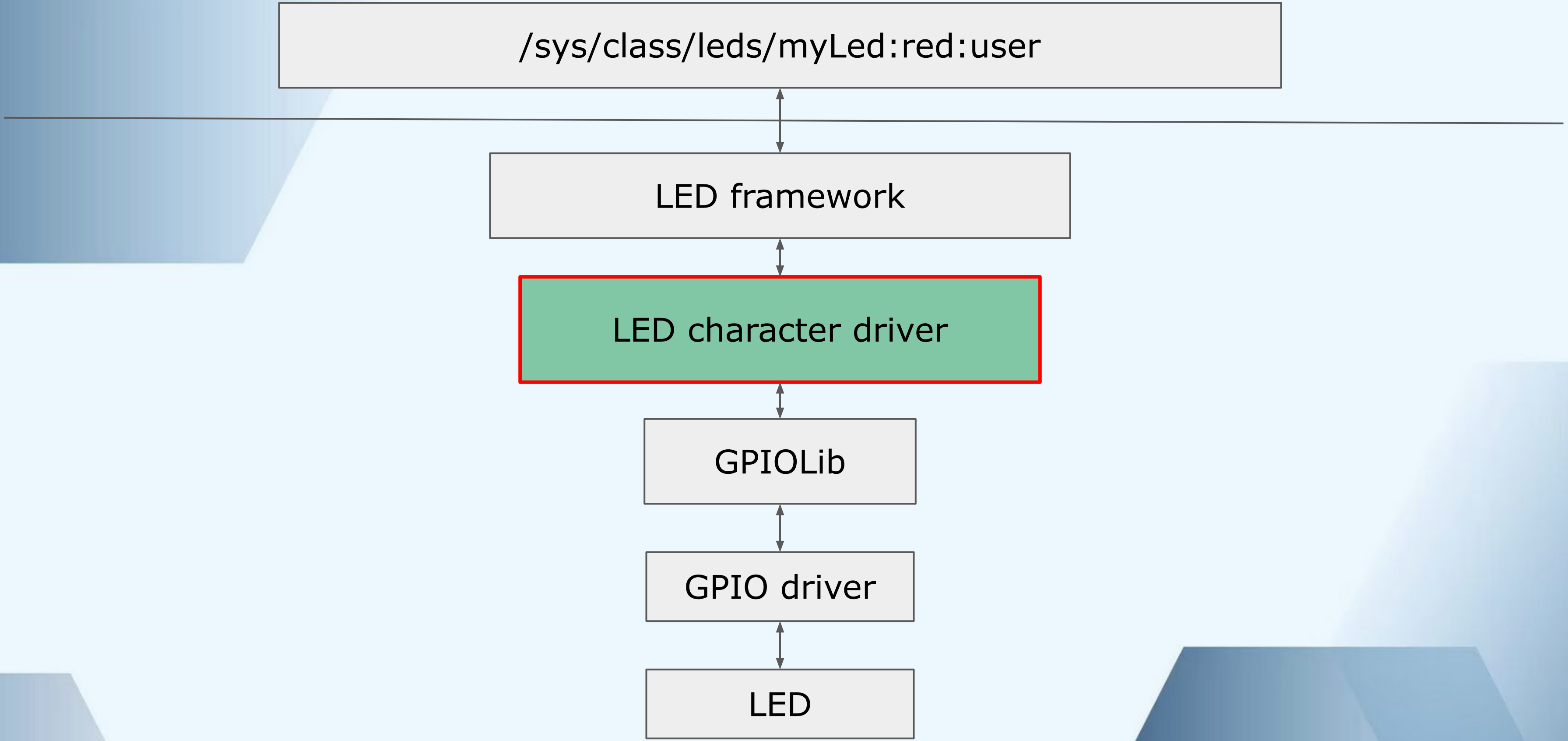
Registering the LED

```
static inline int led_classdev_register(struct device
*parent, struct led_classdev *led_cdev)
```

LED Class Driver



LED Class Driver



Kernel Bus Infrastructure

- Bus core
  - API implementation for a bus, represented by the bus_type struct
- Bus adapters
  - Controller driver for a bus, represented by a device_driver struct
- Bus drivers
  - Driver for a device connected to the bus, represented with a device_driver struct
- Bus devices
  - Representation of a device connected to the bus, represented with a device struct

Kernel Bus Infrastructure

Creating a platform device driver

```
static const struct of_device_id of_myled_match[] = {
    { .compatible = "breadboard,myled" },
    {},};
static struct platform_driver myled_driver = { .driver = {
       .name     = "myleds",
       .owner     = THIS_MODULE,
       .of_match_table = of_myled_match, },
    .probe         = myled_driver_probe,
    .remove        = myled_driver_remove,
};
module_platform_driver(myled_driver);
```

Kernel Bus Infrastructure

Creating a platform device driver

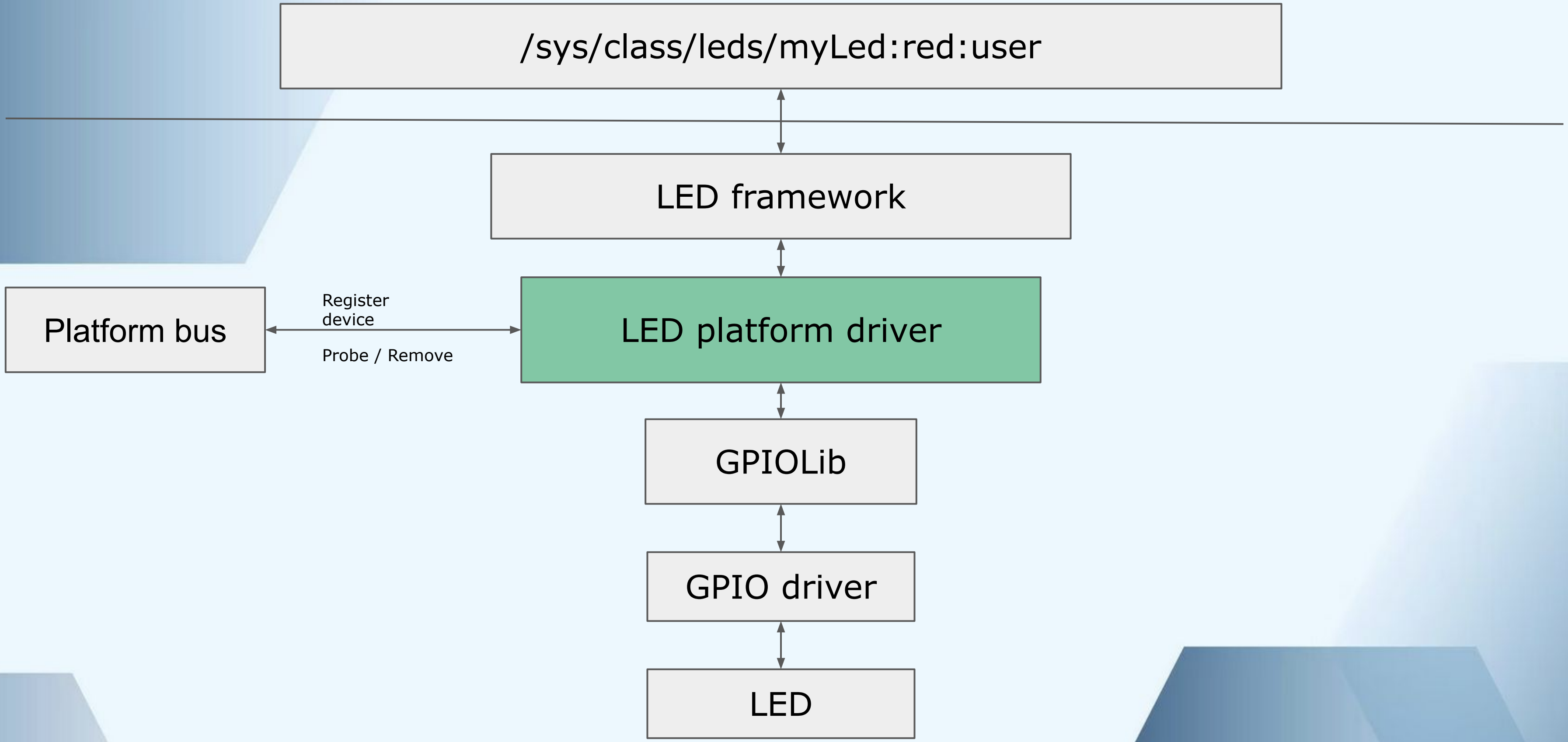
```
static int __init myled_driver_init(void)
static int myled_driver_probe(struct platform_device *pdev)


static void __exit myled_driver_exit(void)
static int myled_driver_remove(struct platform_device *pdev)
```

Kernel Bus Infrastructure

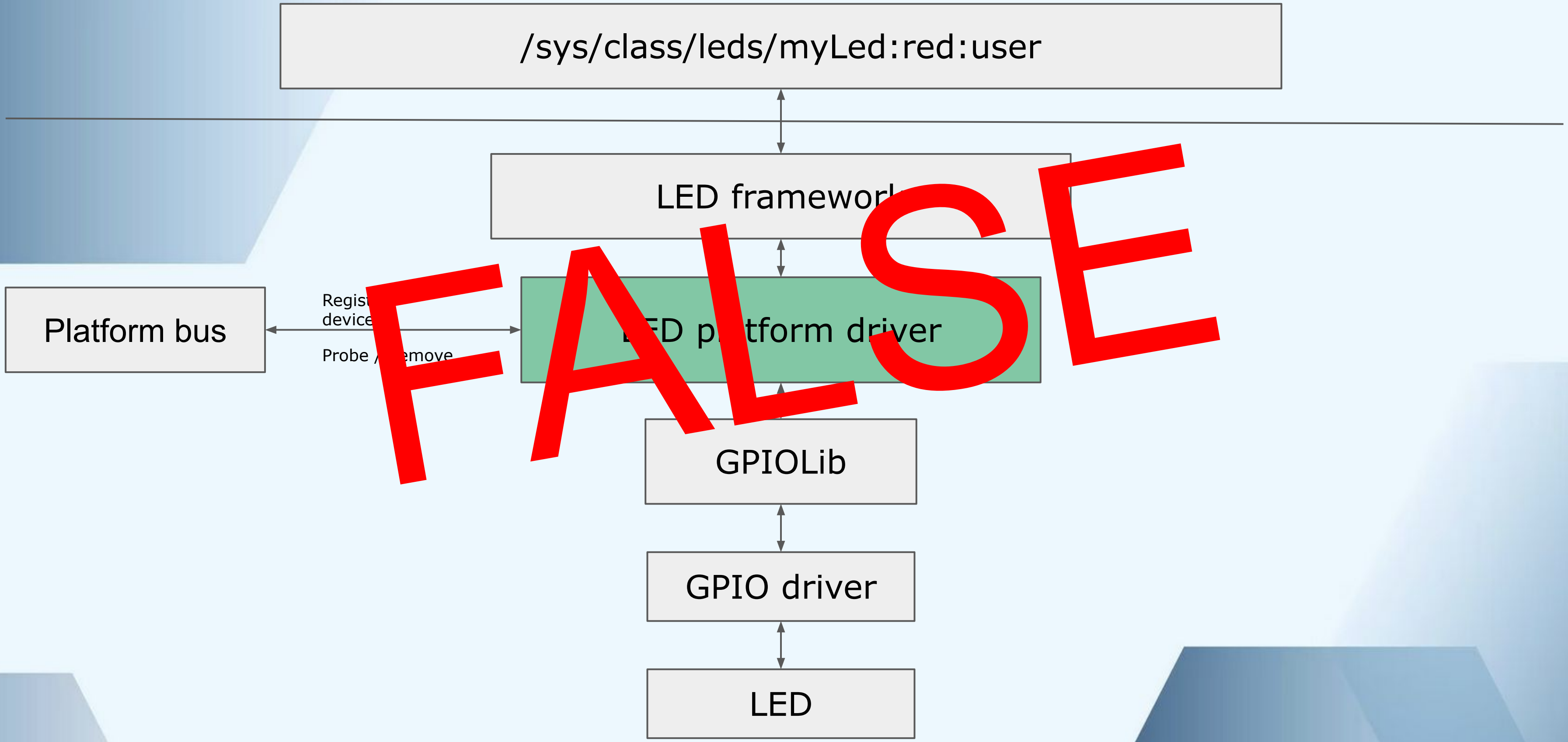
Getting information from platform_device

```
struct gpio_desc *gpiod_get(struct device *dev, const char
*con_id, enum gpiod_flags flags)

struct device_node *of_get_next_child(const struct
device_node *node, struct device_node *prev)

int of_property_read_string(const struct device_node *np,
const char *propname, const char **output)
```

Kernel Bus Infrastructure



Kernel Bus Infrastructure



Hardware description

- Hardware described in the code
  - Hardware registration can be hardcoded in the modules using the function like plaform_device_register()
- Hardware described by a platform mechanism
  - There could be a chip or system in the board that provides information about the hardware, an example of this is ACPI for x86
- Hardware auto discovered
  - There are some buses with enumeration mechanisms that allow them to register the connected hardware based on the bus information, example USD and PCI
- Hardware described in a device tree structure
  - There are devices that cannot provide additional mechanisms to describe its hardware and also its hardware could change depending on the designers, then those changes can be described in device tree sources

Hardware description

What is the device tree?

Device tree is a data structure describing the hardware components of a particular computer so that the operating system's kernel can use and manage those components, including the CPU or CPUs, the memory, the buses and the integrated peripherals.

Device tree sources (dts) are compiled to generate device tree binaries (dtb) with a device tree compiler (dtc), this device tree binaries are the files that are going to be passed to the kernel.

Hardware description

Where are the device tree sources?

In the Linux kernel source code we can find device tree sources in

```
arch/$ARCH/boot/dts/
```

Replacing `$ARCH` with the board architecture

And we can compile them using the target dtbs

```
make ARCH=arm64 CROSS_COMPILE=aarch64-none-linux-gnu- dtbs
```

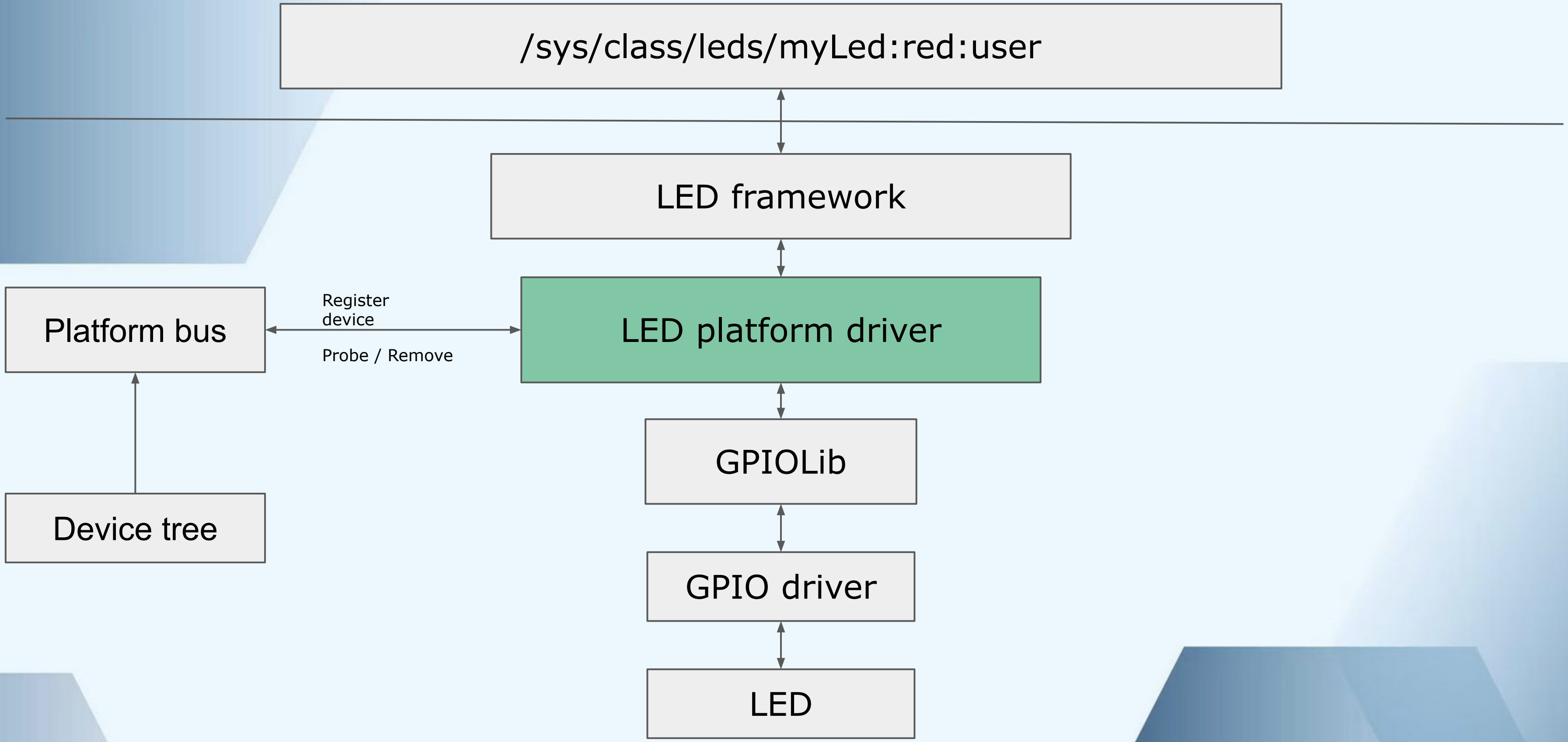
Hardware description

We need to find the dts file corresponding to our board and edit it.

Inside the structure identified as root with an “/”

```
myled {

        compatible = "breadboard,myled";

        label = "myled:red:user";

        myled-gpios = <&gpio 21 GPIO_ACTIVE_LOW>;

};
```

Kernel Bus Infrastructure



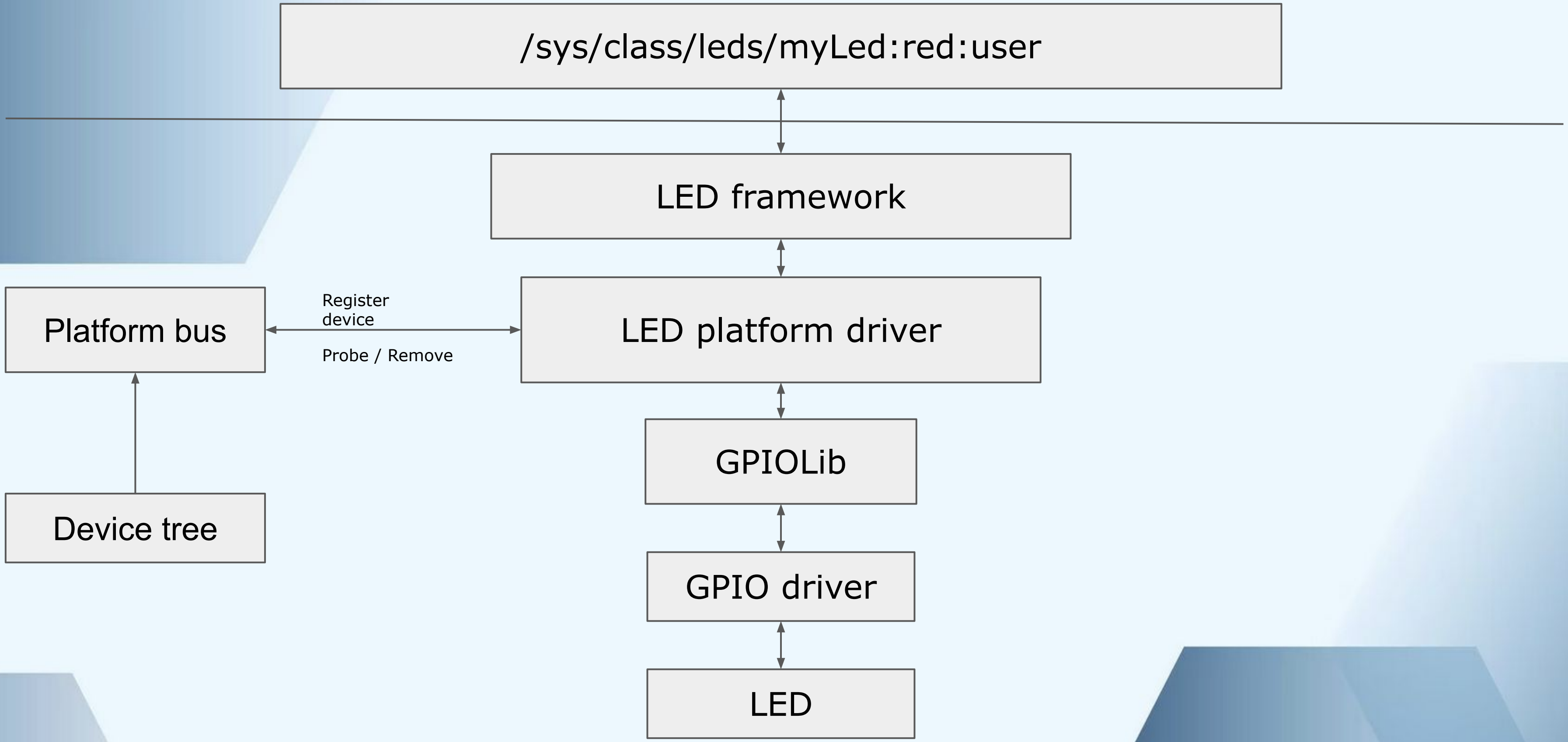
Hardware description

Taking advantage of the available kernel drivers.

In the same file that we edited we can add.

```
&leds {

    compatible = "gpio-leds";

    ...

    myled {

        label = "myled:red:user";

        gpios = <&rp1_gpio 21 GPIO_ACTIVE_LOW>;

    };

};
```

Kernel Bus Infrastructure



References

Ben Klemens, 21st Century C

Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman, Linux Device Drivers

https://docs.kernel.org/

https://www.raspberrypi.com/documentation

https://elixir.bootlin.com/linux/v6.6.21/source

Sergio Prado, 2021, Understanding the Structure of a Linux Kernel Device Driver

Thank you

Contact:
hernandez.josued@gmail.com
https://github.com/josuedhg
josuedhg | josue_dhg on internet
Acruth for videogames

Appendix

Custom Kernel Driver

Custom kernel compilation

- `yay -Sy git bc bison flex libssl-dev make aarch64-none-linux-gnu-gcc-12.3-bin`
- `git clone --depth=1 https://github.com/raspberrypi/linux && cd linux`
- `make ARCH=arm64 CROSS_COMPILE=aarch64-none-linux-gnu- bcm2712_defconfig`
- `make ARCH=arm64 CROSS_COMPILE=aarch64-none-linux-gnu- Image modules dtbs`

Custom Kernel Driver

Installing custom kernel

- `mkdir -p mnt/fat32 && mkdir mnt/ext4`
- `sudo mount /dev/sdx1 mnt/fat32 && sudo mount /dev/sdx2 mnt/ext4`
- `sudo make ARCH=arm64 CROSS_COMPILE=aarch64-none-linux-gnu- INSTALL_MOD_PATH=mnt/ext4 modules_install`
- `sudo cp arch/arm/boot/zImage mnt/fat32/kernel_2712.img`
- `sudo cp arch/arm64/boot/dts/broadcom/*.dtb mnt/fat32/`
- `sudo cp arch/arm64/boot/dts/overlays/*.dtb* mnt/fat32/overlays/`
- `sudo umount mnt/fat32`
- `sudo umount mnt/ext4`