Embedded Linux Conference 2021

Understanding the structure of a Linux kernel device driver

Embedded **Labworks**

By Sergio Prado. São Paulo, September 2021

$ WHOAMI

- Embedded software developer for more than 25 years.
- Consultant and trainer at Embedded Labworks.
  https://e-labworks.com/en
- Open source contributor (Buildroot, Yocto Project, Linux kernel, etc).
- Blogger at https://embeddedbits.org/.

ABOUT THE TALK

- The objective of this talk is to cover the modern architecture of a Linux kernel device driver.
- This is an entry level talk about the main pieces of a device driver in the Linux kernel (let's connect the dots!).
- We will not cover Linux driver model's internal structures and data representation (`kobjects`, `ktypes`, `ksets`, etc).
- We will also not cover Linux API's in details.
- In the end of this talk, you will be able to read and understand a modern Linux device driver, and why not, start writing your own!

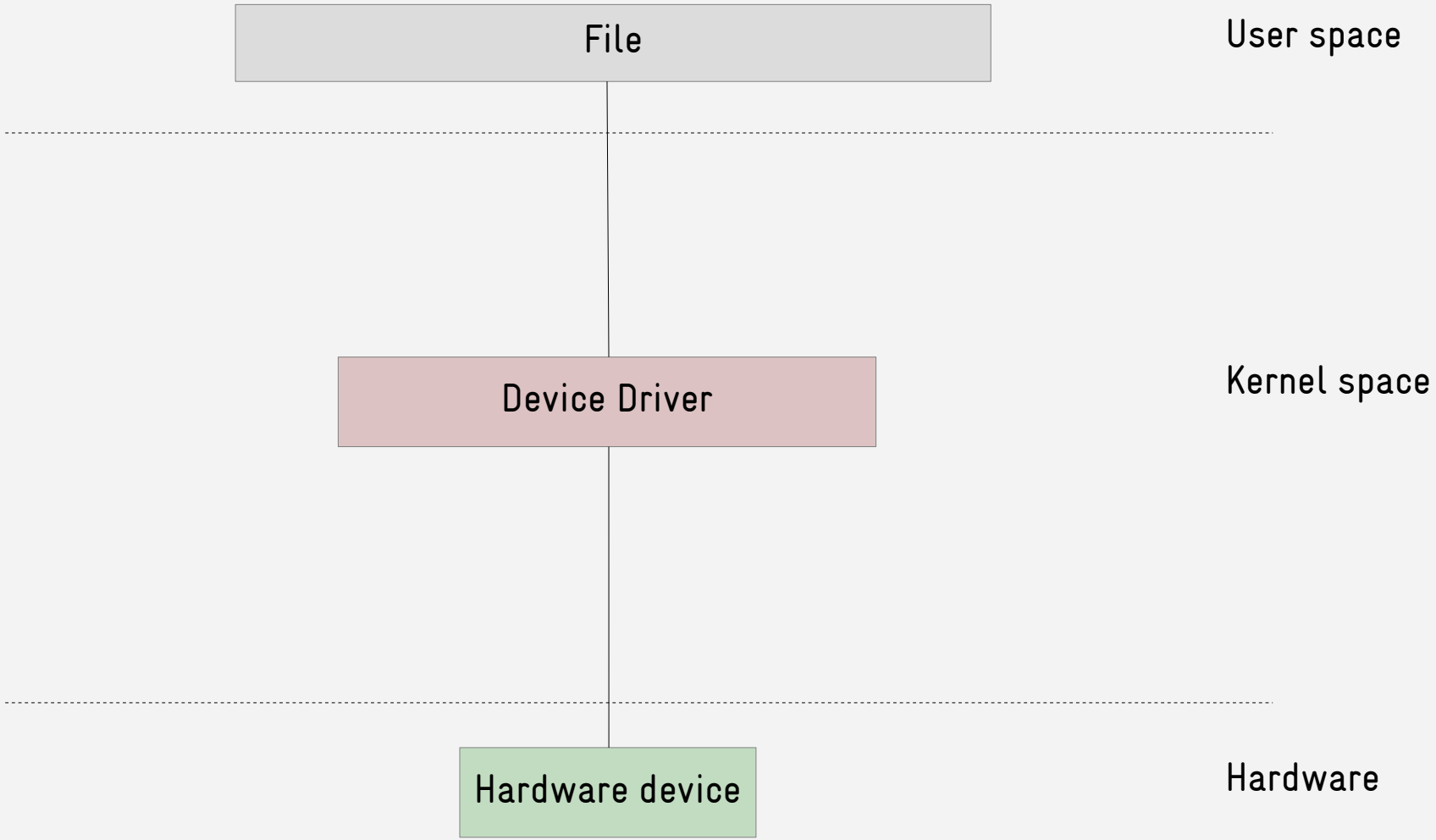
AGENDA

1. Introduction to device drivers and character drivers
2. Hardware access (MMIO, gpiolib)
3. Introduction to the driver model
4. Frameworks
5. Buses and device tree
6. Lots of hands-on!

WHAT ARE DEVICE DRIVERS?

- Device drivers are just abstractions to a piece of hardware!
- Although you can have device drivers running in userspace (via some kernel interface like UIO or I2CDEV), it's much more common to have them running in kernel space.
- Providing an infrastructure to write and run device drivers is part of an operating system's kernel responsibility.
- On Unix based systems, a file is the most common abstraction to communicate with a device.

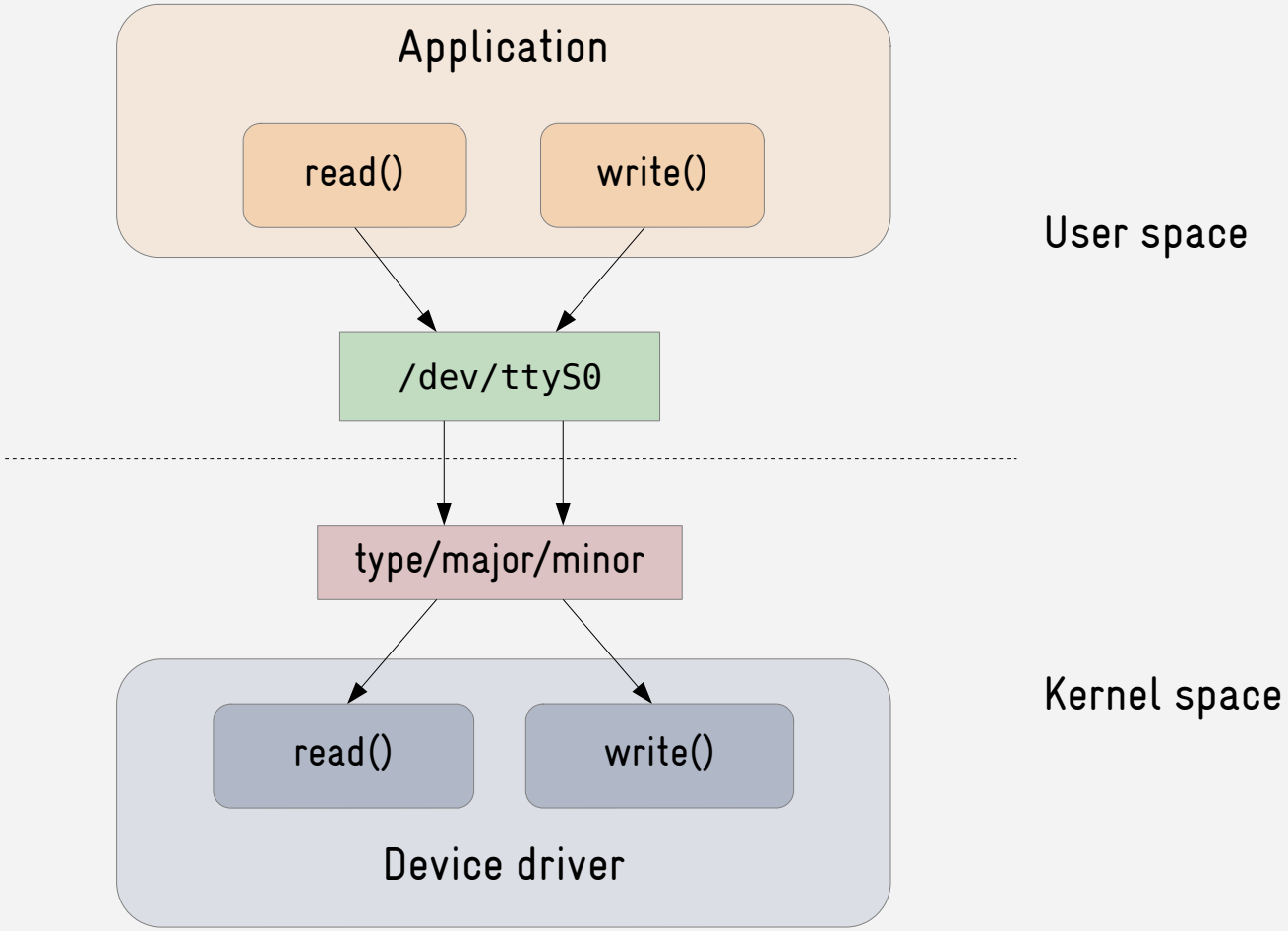
DEVICE DRIVER IS AN ABSTRACTION



CHAR DRIVER: A SIMPLE ABSTRACTION

- There are a few kernel APIs that could be used to export an interface to userspace in the filesystem (`/dev`, `/sys`, etc).
- Device nodes are one of the interfaces that could be used, where files are exported to users in `/dev` in the form of **char** or **block** device files.
- These device files have associated three basic information:
  - Type (block or char).
  - Major number.
  - Minor number.

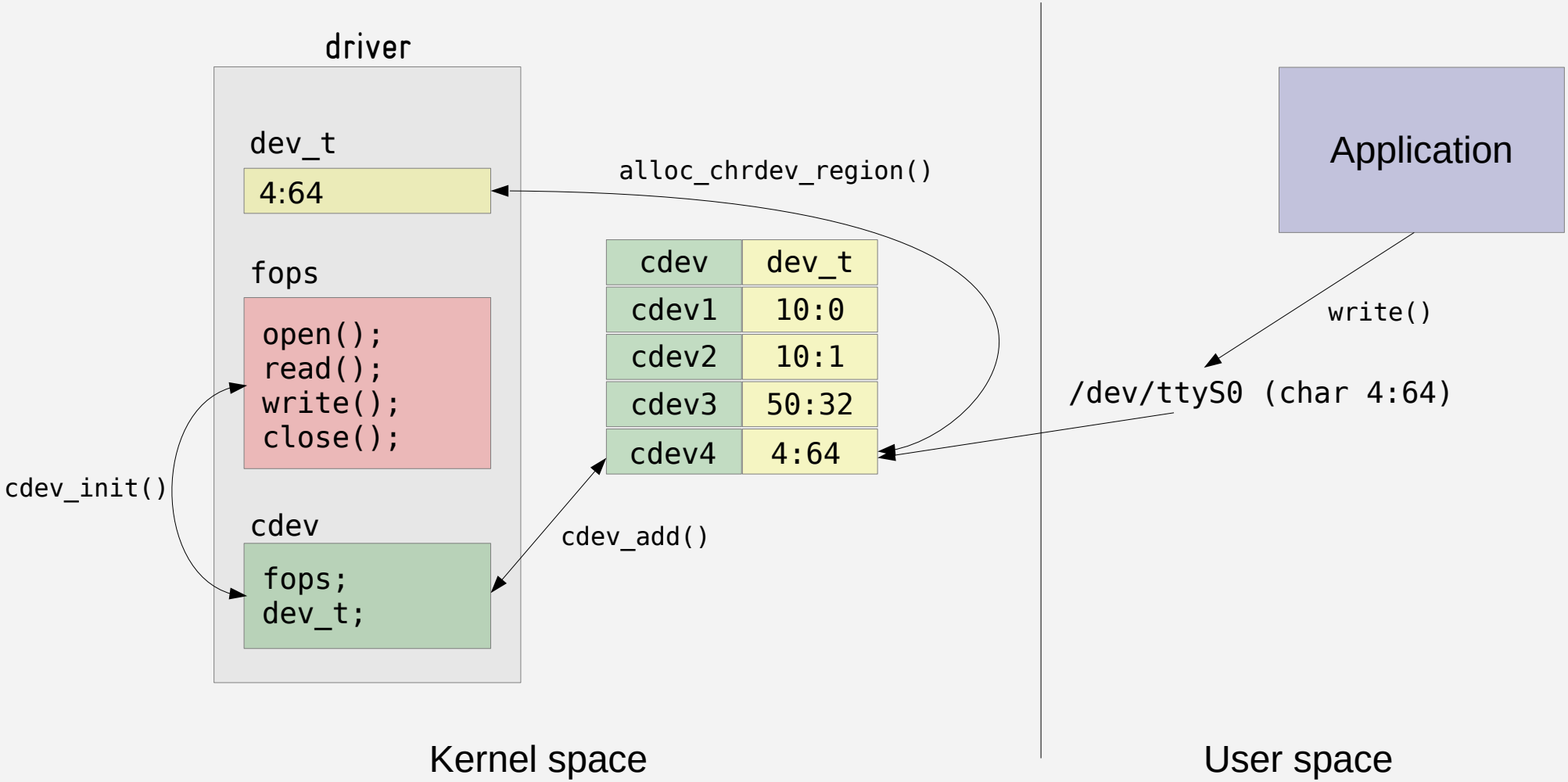
CHAR DRIVER AS A FILE ABSTRACTION



IMPLEMENTING A CHAR DRIVER

- **Step 1**: Allocate the device number (major/minor). This can be done with `register_chrdev_region()` or `alloc_chrdev_region()`.
- **Step 2**: Implement the file operation (open, read, write, ioctl, etc).
- **Step 3**: Register the char driver in the kernel with `cdev_init()` and `cdev_add()`.

CHAR DRIVER

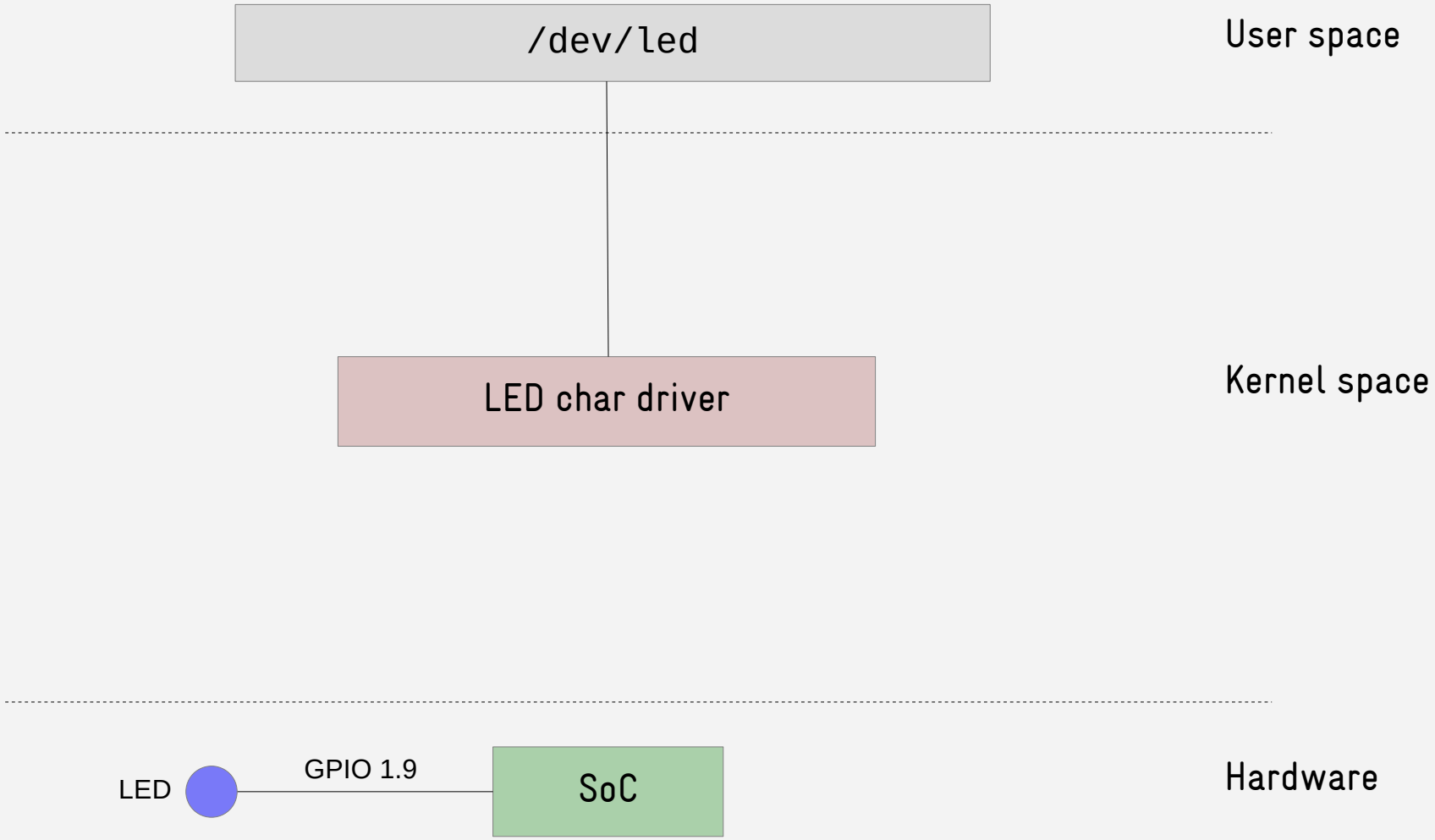

driver
dev_t
4:64
fops
open();
read();
write();
close();
cdev
fops;
dev_t;
cdev_init()
alloc_chrdev_region()
cdev_add()
cdev
dev_t
cdev1
10:0
cdev2
10:1
cdev3
50:32
cdev4
4:64
Application
write()
/dev/ttyS0 (char 4:64)
Kernel space
User space

Embedded Labworks

Hands-on 1

Writing a char driver

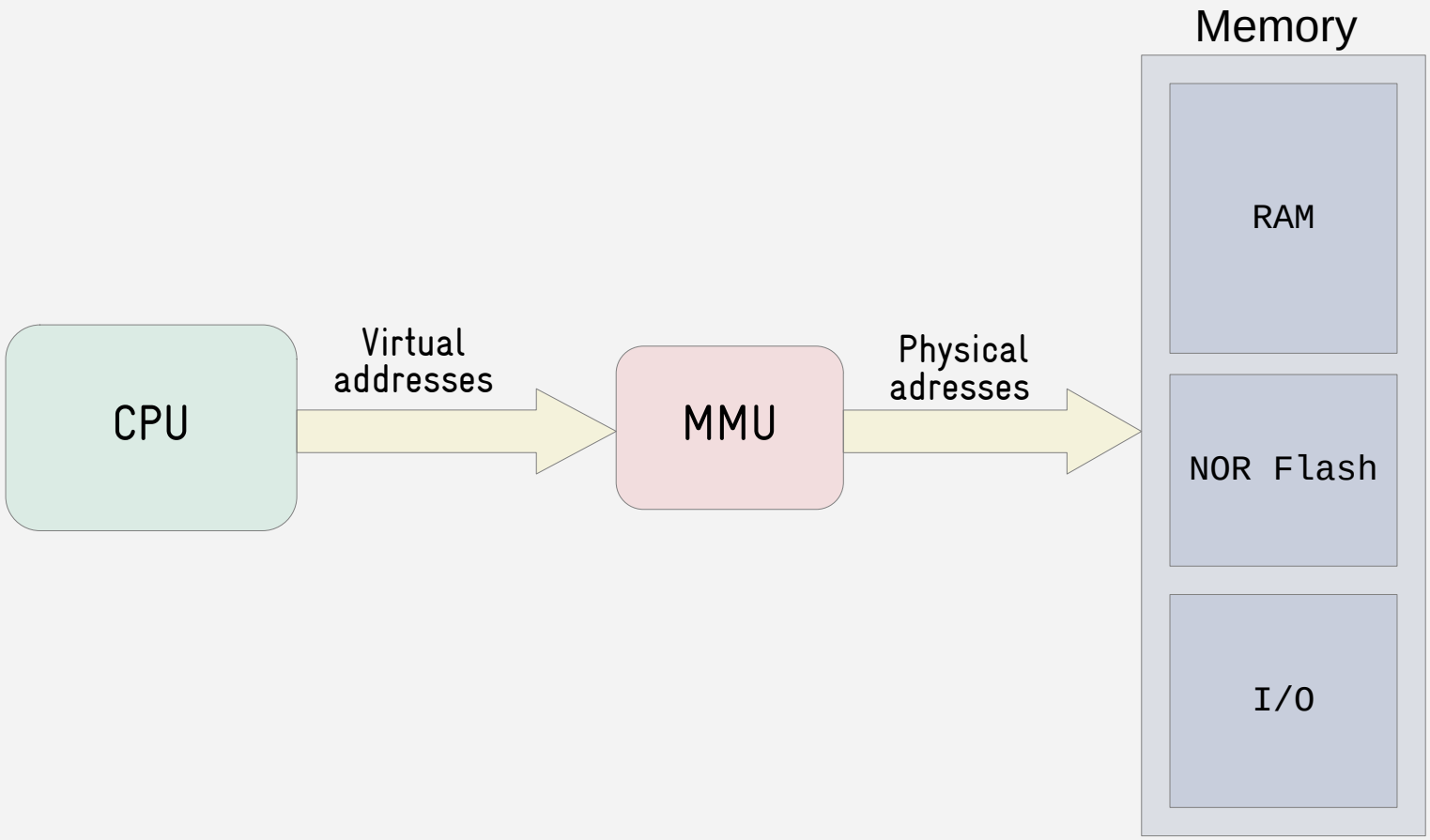
LED DRIVER


/dev/led
User space
LED char driver
Kernel space
LED
GPIO 1.9
SoC
Hardware

TALKING TO THE HARDWARE

- Depending on the hardware architecture, there are a few mechanisms a CPU can use to communicate with a hardware device, including:
  - **Port I/O**: a dedicated bus is used to communicate with hardware devices.
  - **Memory-mapped I/O**: the memory address space is shared with hardware devices.
- MMIO is currently the most common approach adopted by popular architectures like ARM.

MEMORY-MAPPED I/O



TALKING TO A MMIO DEVICE

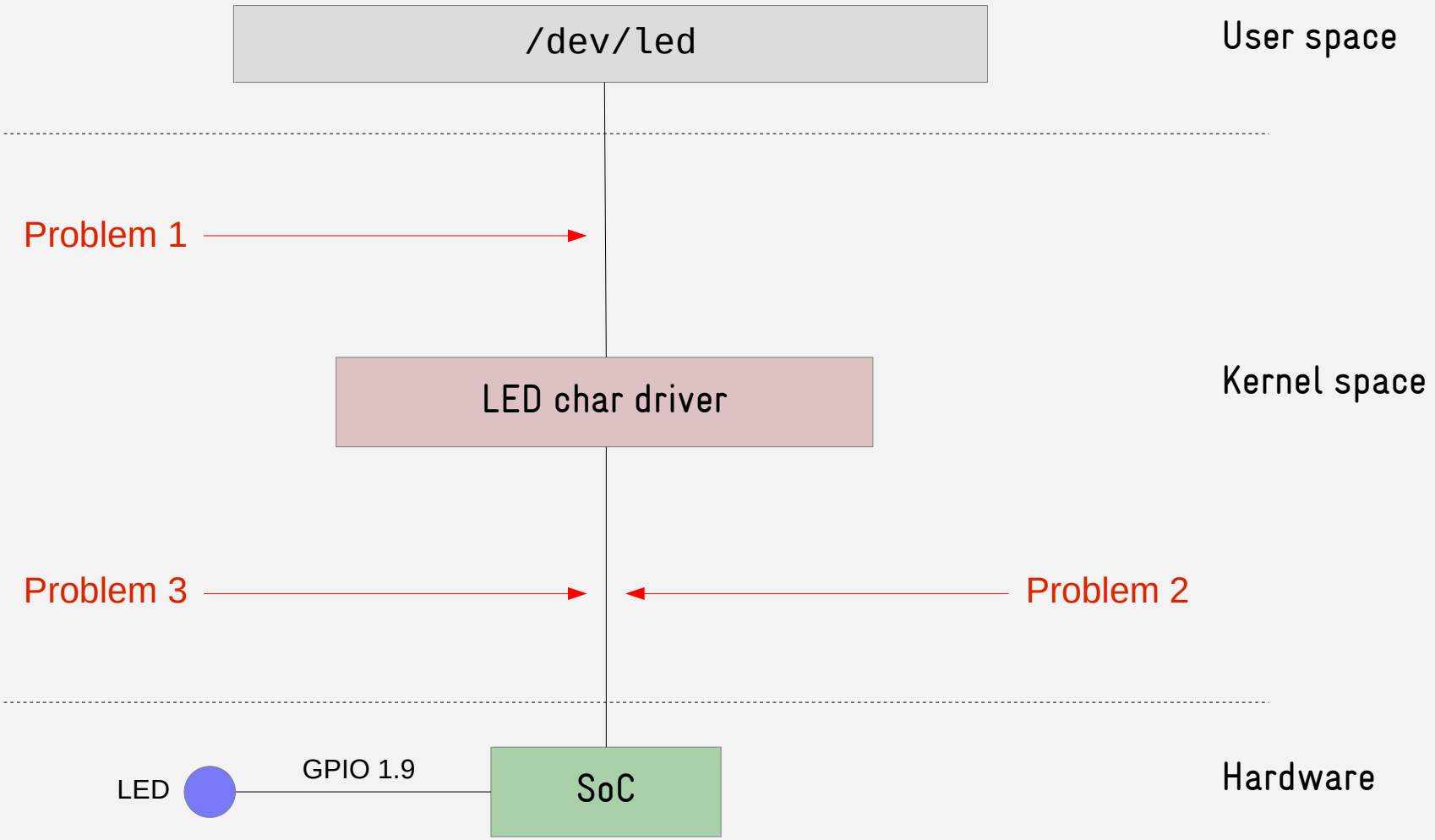
- **Step 1**: request access to MMIO registers using a few kernel APIs like `request_mem_region()`.
- **Step 2**: Map the registers' physical addresses to virtual addresses, using functions like `ioremap()`.
- **Step 3**: Use the kernel API to read from and write to the registers, with functions like `readl()` and `writel()`.

Embedded **Labworks**

Hands-on 2

Talking to a MMIO based device

LED DRIVER

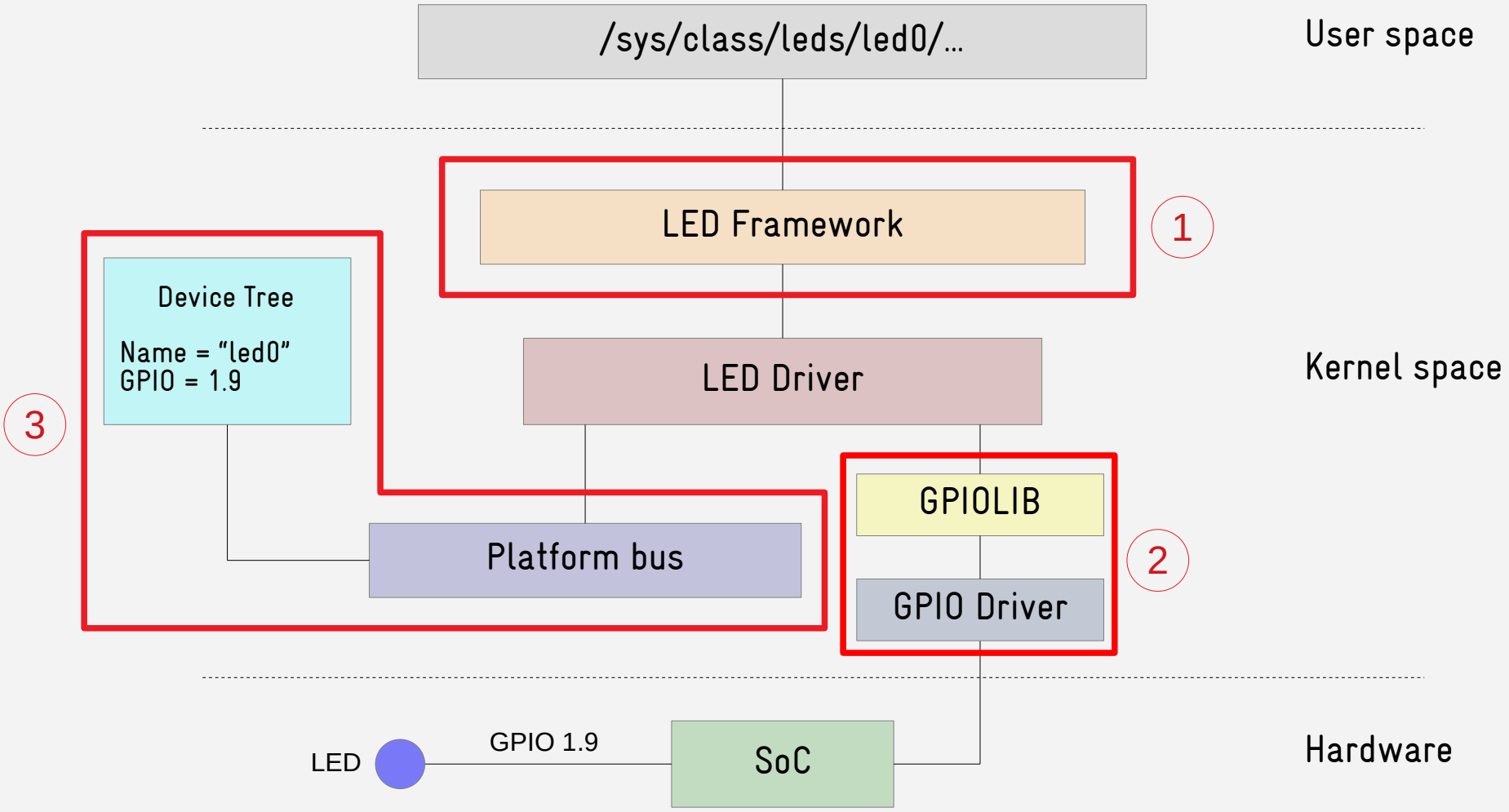

/dev/led
User space
Problem 1
LED char driver
Kernel space
Problem 3
Problem 2
LED
GPIO 1.9
SoC
Hardware

THE DRIVER MODEL

- The Linux driver model provides several abstractions to device drivers to make the code more modular, reusable and easy to maintain.

- Among its components, we have:
  - **Frameworks**: the interface exported by a type or class of devices is standardized.
  - **Buses**: information about the devices and where they are connected is abstracted away from the driver.

LED DRIVER


/sys/class/leds/led0/...
User space
LED Framework
1
Device Tree
Name = "led0"
GPIO = 1.9
3
LED Driver
Kernel space
GPIOLIB
Platform bus
2
GPIO Driver
LED
GPIO 1.9
SoC
Hardware

FRAMEWORKS

- Frameworks make it easier to provide a standardized interface and abstraction for users.
  - Device drivers developers don't need to think about the interface that will be exported to users.
  - Users know beforehand the interface provided by a driver based on its class or type.
- There are frameworks for the most common types of devices: input, IIO, ALSA, V2L2, RTC, watchdog, etc!
- And of course, this is a framework for LED drivers!

USING THE LEDS FRAMEWORK

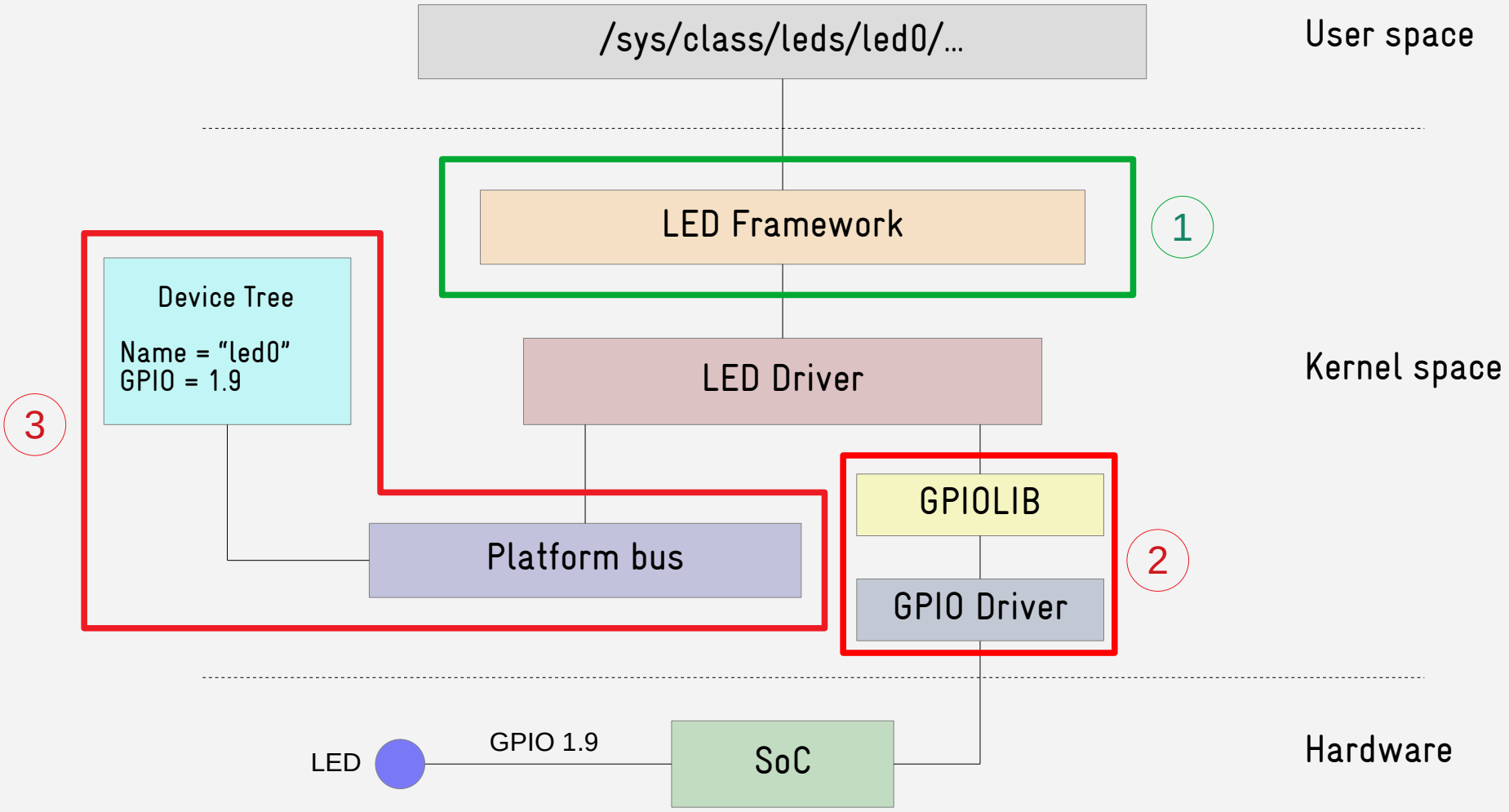
- **Step 1**: Initialize an structure of type `led_classdev`.
- **Step 2**: Provide (at least) a callback function to change the status of the LED.
- **Step 3**: Register the driver in the framework with the function `led_classdev_register()`.

Embedded Labworks

Hands-on 3

Using the LEDS framework

LED DRIVER


/sys/class/leds/led0/...
User space
LED Framework
1
Device Tree
Name = "led0"
GPIO = 1.9
3
LED Driver
Kernel space
GPIOLIB
Platform bus
2
GPIO Driver
LED
GPIO 1.9
SoC
Hardware

GPIOLIB

- The Linux kernel implements a **producer/consumer** model for GPIO management:
  - There are GPIO "producers" like GPIO controllers drivers.
  - There are GPIO "consumers" like a LED driver, keyboard driver or a userspace application.
- **gpiolib** is the API provided by the Linux kernel for GPIO producers and consumers.

GPIOLIB API

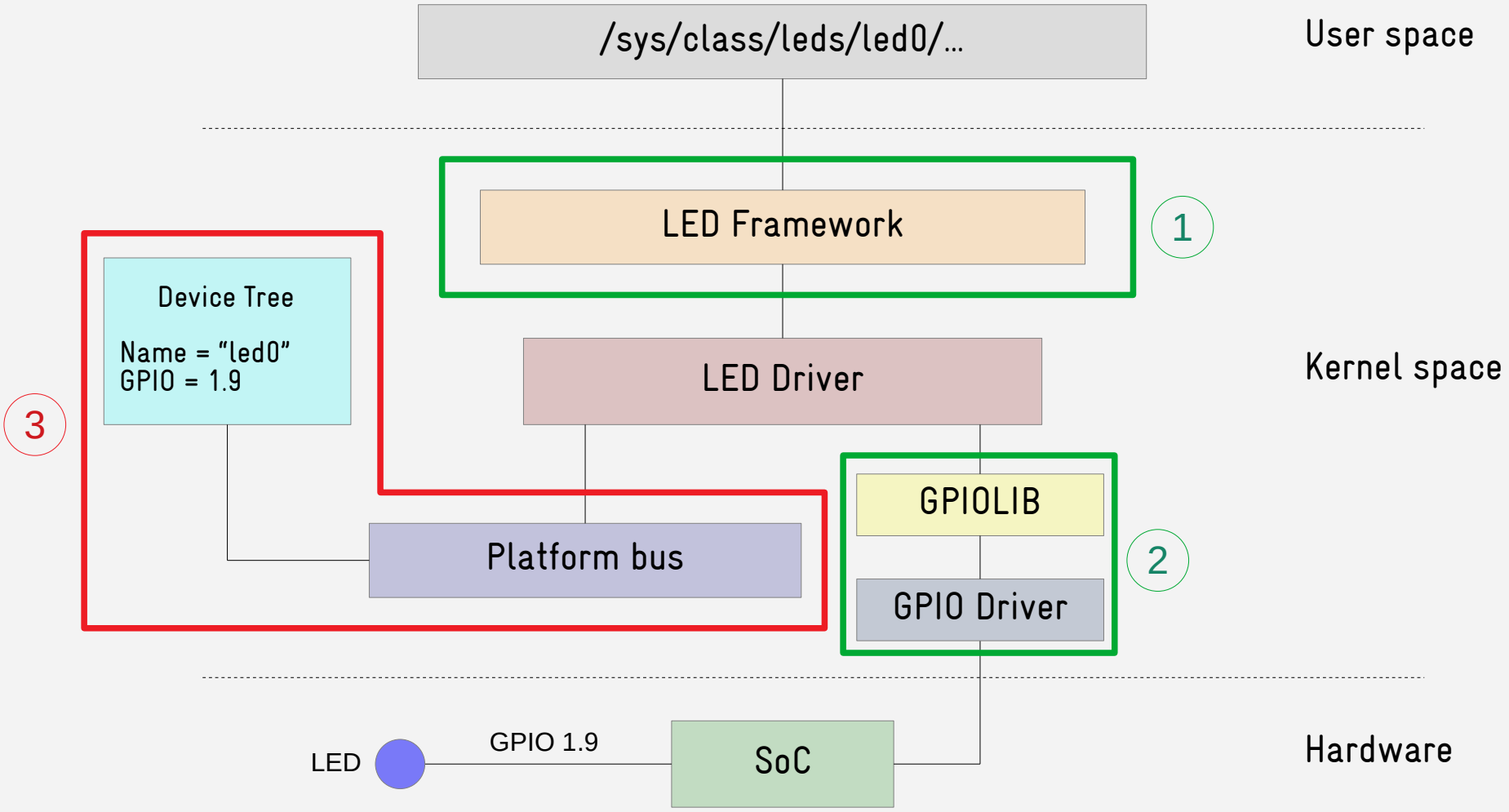
```
#include <linux/gpio.h>
#include <linux/gpio/consumer.h>

struct gpio_desc *gpiod_get(struct device *dev,
                            const char *con_id,
                            enum gpiod_flags flags);
void gpiod_put(struct gpio_desc *desc);

int  gpiod_direction_input(struct gpio_desc *desc);
int  gpiod_direction_output(struct gpio_desc *desc, int value);

void gpiod_set_value(struct gpio_desc *desc, int value);
int  gpiod_get_value(const struct gpio_desc *desc);

struct gpio_desc *gpio_to_desc(unsigned gpio);
int  desc_to_gpio(const struct gpio_desc *desc);

int  gpiod_to_irq(const struct gpio_desc *desc);
```

Embedded Labworks

Hands-on 4

Accessing the hardware via GPIOLIB

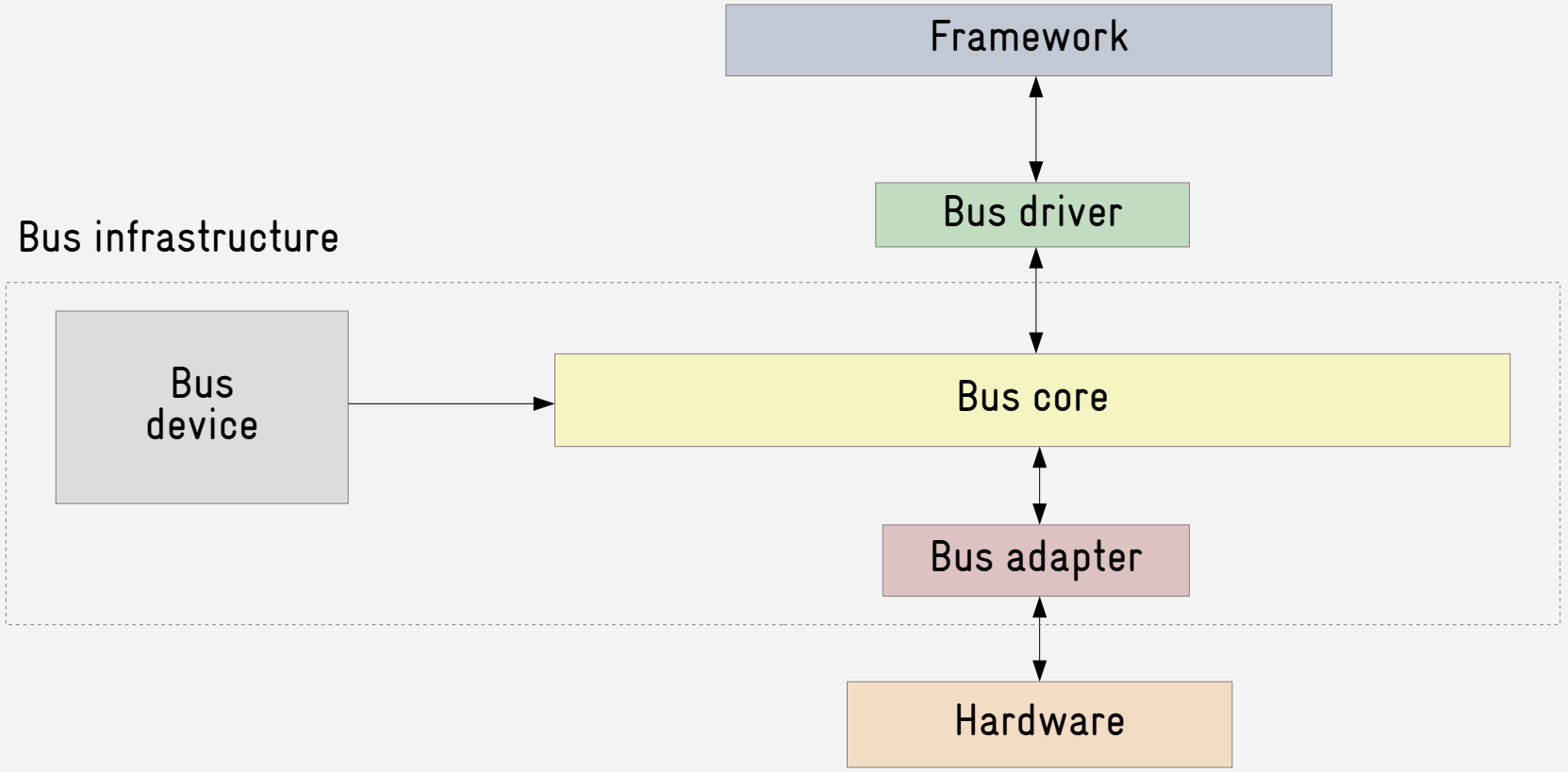
LED DRIVER


/sys/class/leds/led0/...
User space
LED Framework
1
Device Tree
Name = "led0"
GPIO = 1.9
3
LED Driver
Kernel space
GPIOLIB
Platform bus
2
GPIO Driver
LED
GPIO 1.9
SoC
Hardware

BUS INFRASTRUCTURE

- **Bus core**: API implementation for a given bus (USB core, SPI core, I2C core, PCI core, etc), represented in the kernel by the `bus_type` structure.
- **Bus adapters**: bus controller drivers, represented in the kernel by the `device_driver` structure.
- **Bus drivers**: drivers responsible for managing a device connected to the bus, represented in the kernel by the `device_driver` structure.
- **Bus devices**: devices connected to the bus, represented in the kernel by the structure `device`.

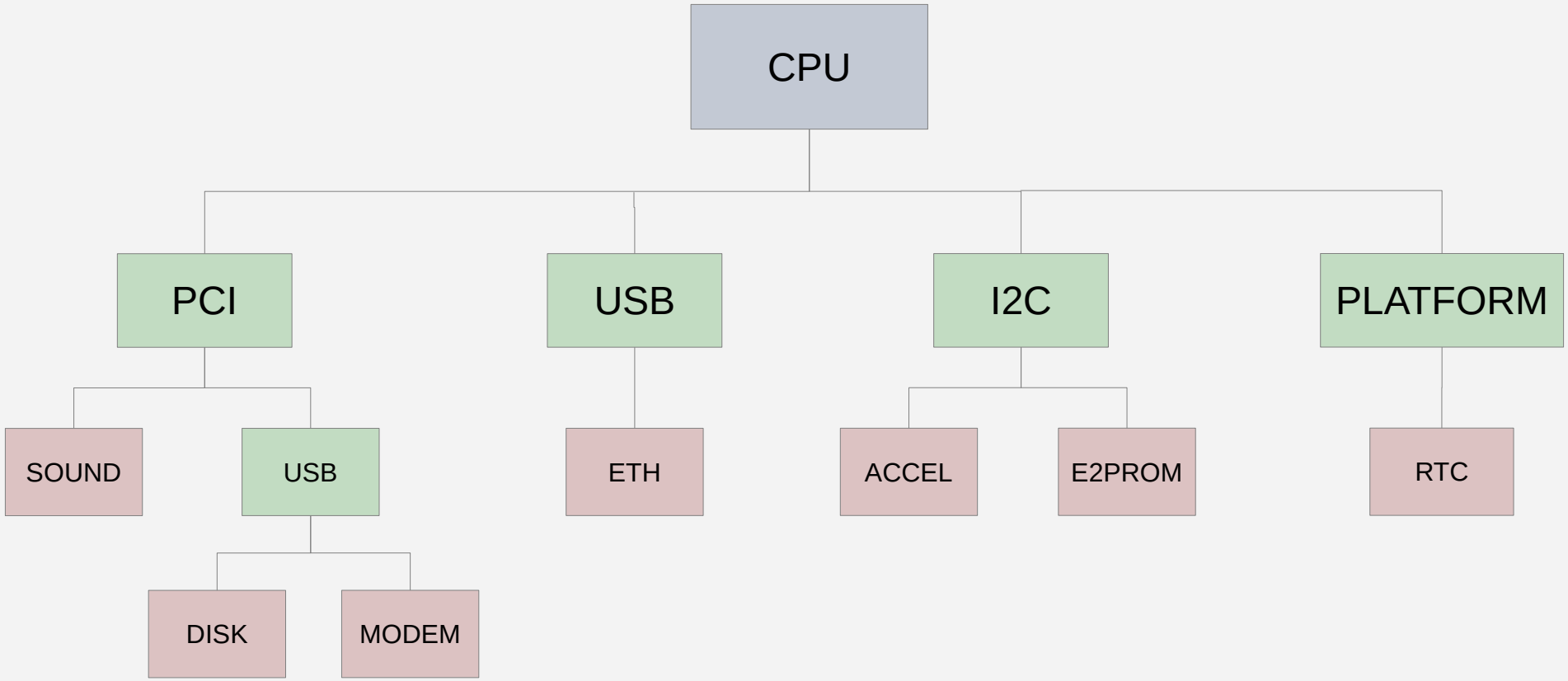
BUS INFRASTRUCTURE (cont.)



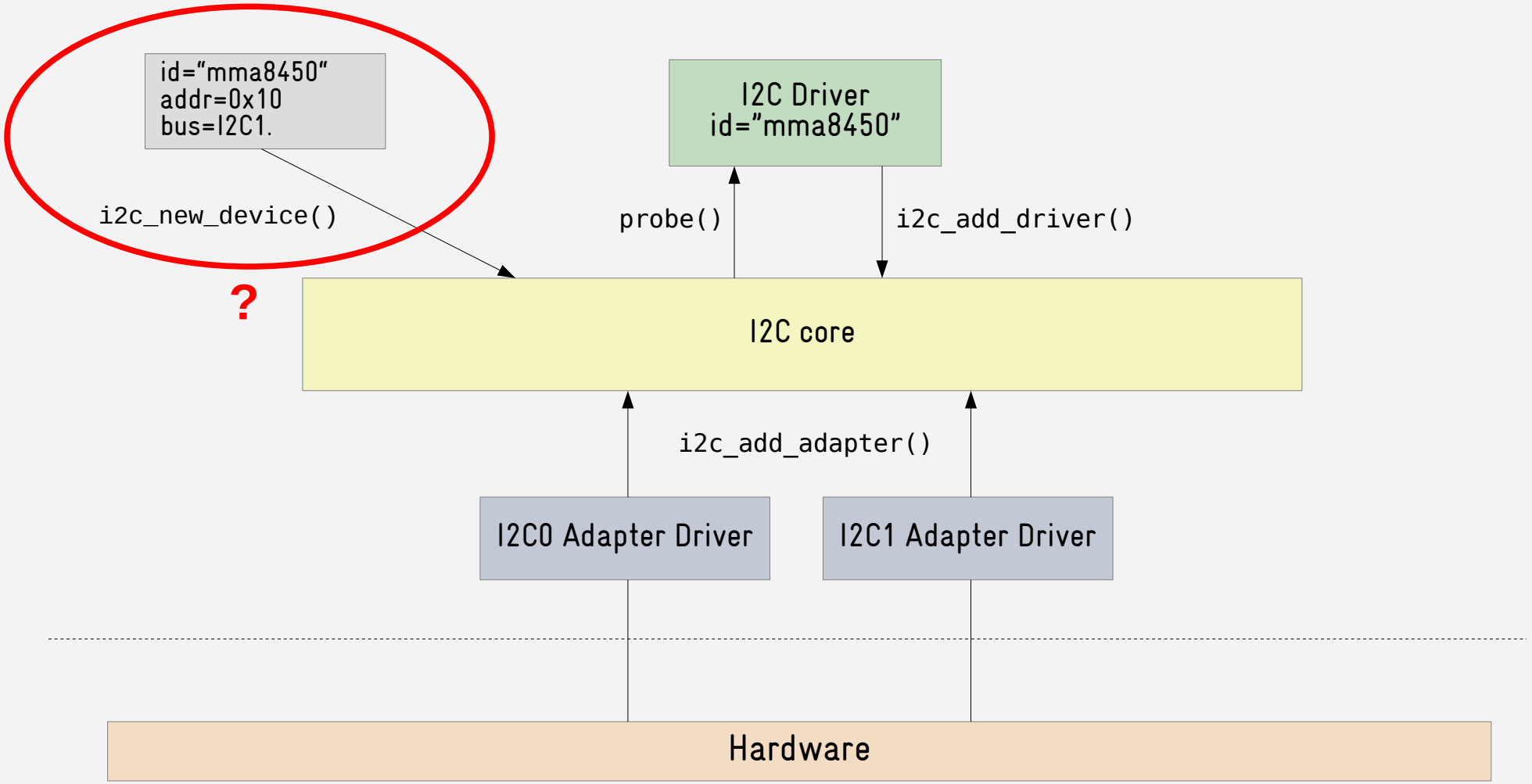
ADVANTAGES

- More control over the access on a given bus.
- Separate the driver code from the device description.
- Easier to identify the hierarchy of devices and buses.
- Improves power management.
- Improves usability and modularity.

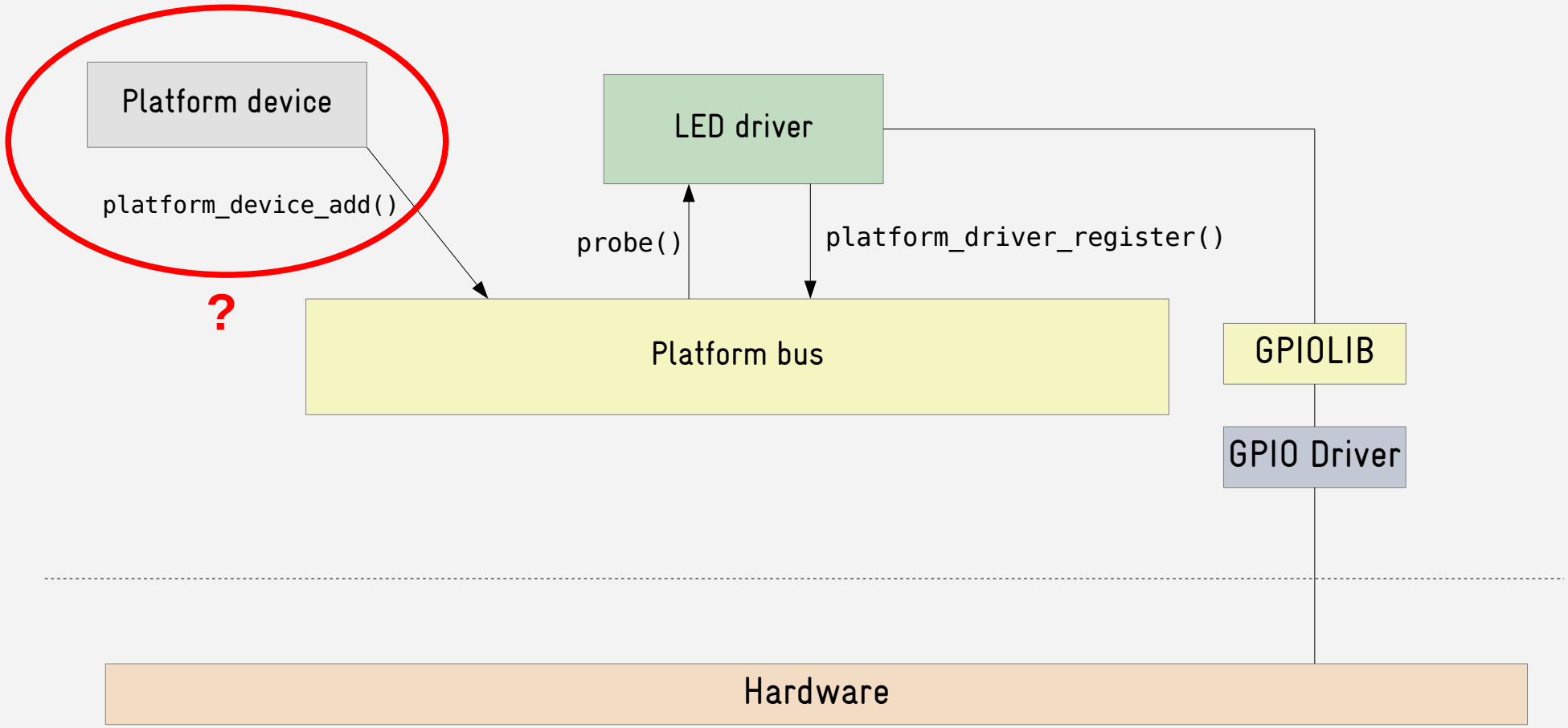
BUSES AND POWER MANAGEMENT



I2C BUS


id="mma8450"
addr=0x10
bus=I2C1.
i2c_new_device()
?
I2C Driver
id="mma8450"
probe()
i2c_add_driver()
I2C core
i2c_add_adapter()
I2C0 Adapter Driver
I2C1 Adapter Driver
Hardware

PLATFORM BUS



REGISTERING A DEVICE

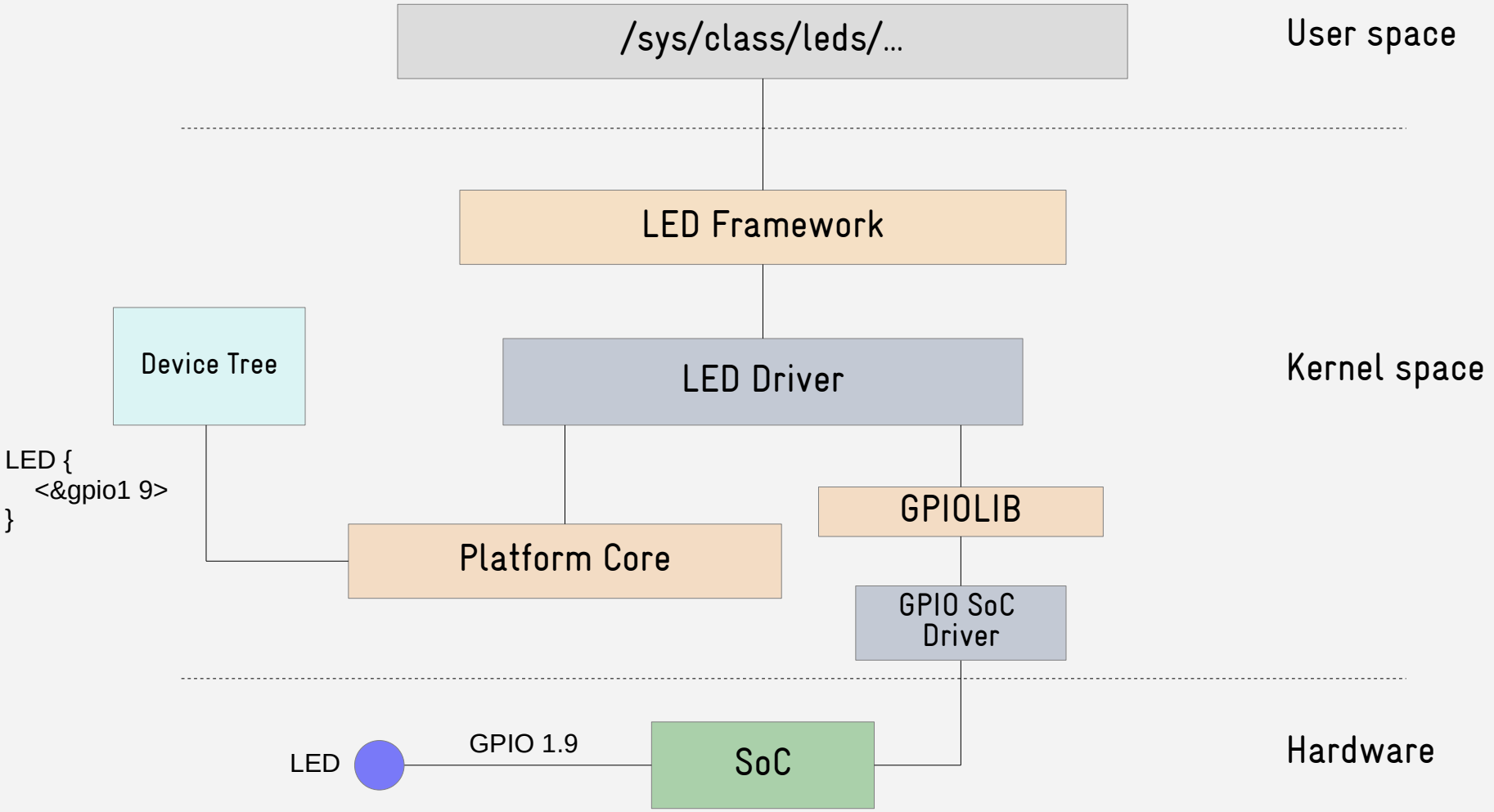
- A device can be statically registered in the code through an API provided by the bus core, such as `i2c_register_board_info()` or `platform_device_register()` (this mechanism is deprecated).
- Devices can be described and registered via a mechanism provided by the hardware platform (e.g. ACPI on x86).
- Devices can be described and registered via a device tree (standard mechanism in some architectures such as PowerPC and ARM).
- Buses that support device enumeration can automatically identify and register devices (e.g. USB and PCI).

Embedded Labworks

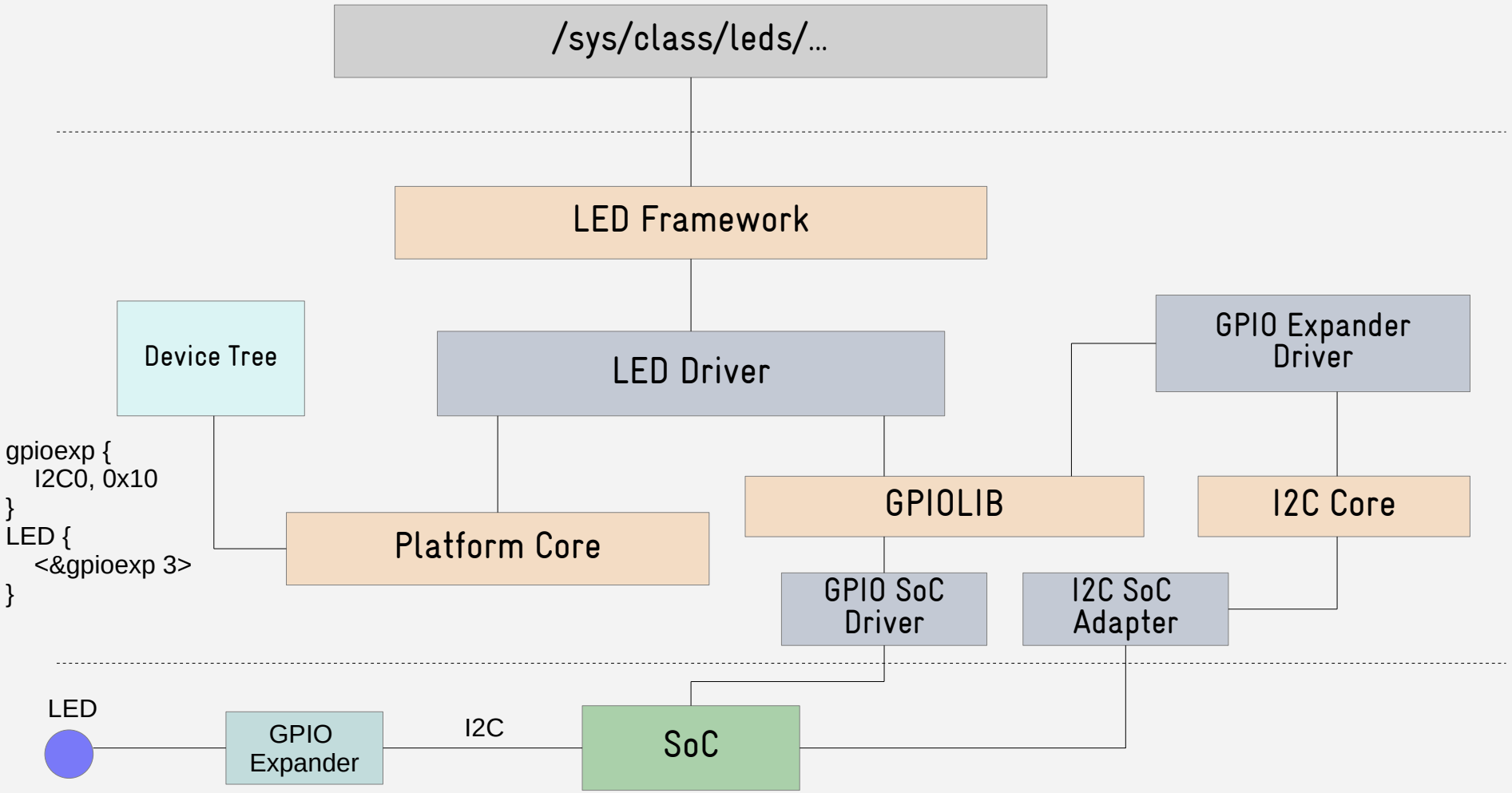
Hands-on 5

Platform bus and device tree

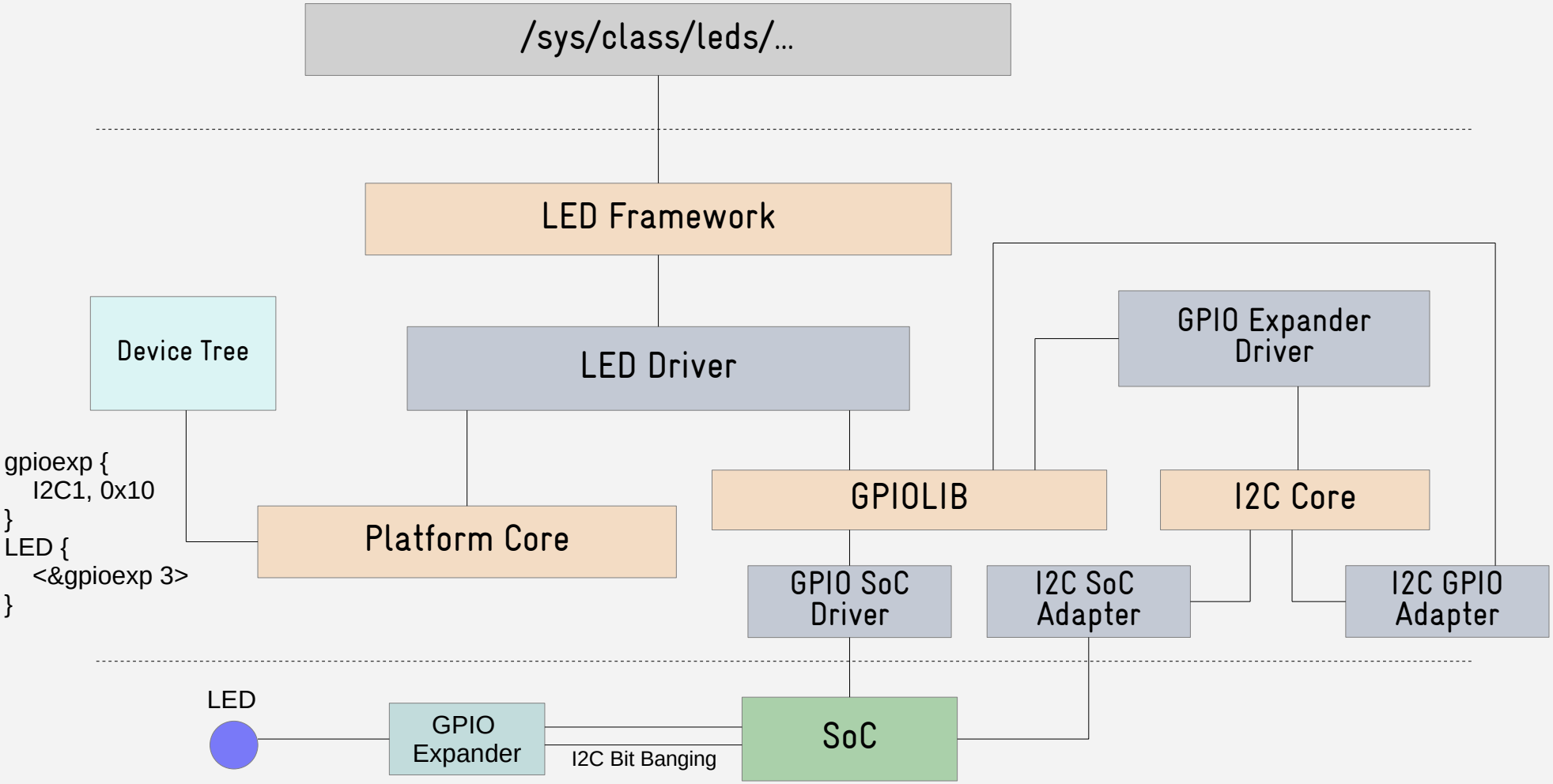
A FLEXIBLE MODEL



A FLEXIBLE MODEL (cont.)



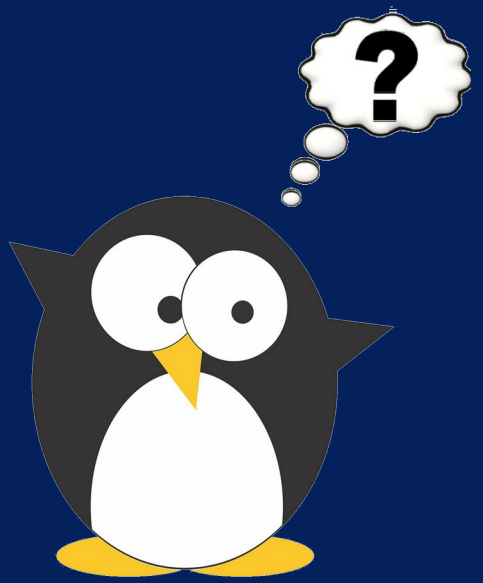
A FLEXIBLE MODEL (cont.)



Thanks!

E-mail sergio.prado@e-labworks.com
Website https://e-labworks.com/en

Linkedin https://www.linkedin.com/in/sprado
Twitter https://twitter.com/sergioprado
Blog https://embeddedbits.org



Embedded **Labworks**

By Sergio Prado. São Paulo, September 2021