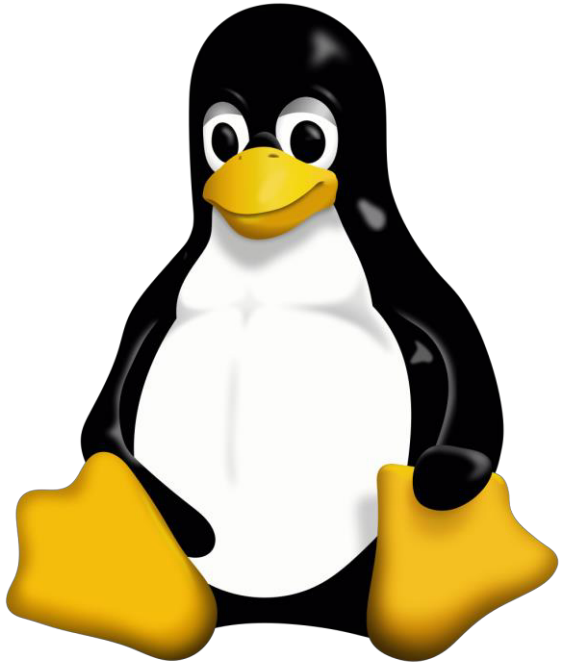


Linux Character Device Driver

Sunbeam Infotech

Kernel thread/daemon

Step1: implement a thread func.

```
int my_func(void *data) {
    for(i=0; i<10; i++) {
        pr_info("%d\n", i);
        msleep(1000);
    }
    return 0;
}
```

Step2: start the thread.

```
kthread_run(my_func, NULL, "th-name");
```

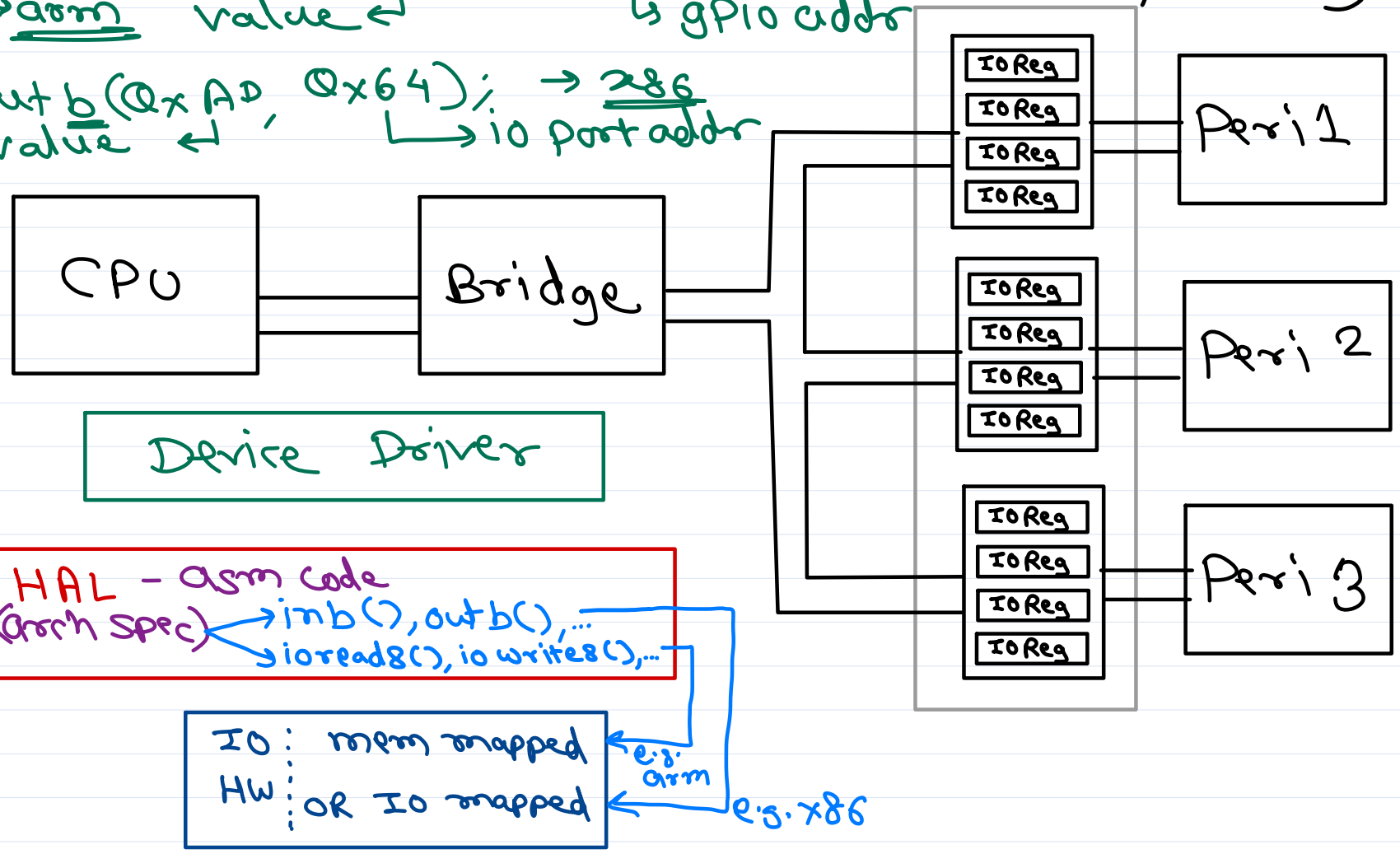
- ① kthread_create() → create thread - sleep state.
- ② wake_up_process() → wakeup thread.

IO Ports

request_region() } io ports | io mem { request_mem_region()
release_region() } io ports | io mem { release_mem_region()

iowrite32(0x55AA55AA, 0x4804c194);
asm value gpio addr

outb(0xAD, 0x64); → x86
value → io port addr



Memory mapped IO

- IO bus & memory bus is overlapping
- Same instn for mem & IO access
- e.g. ARM, ...
  - LDR, STR.

IO mapped IO

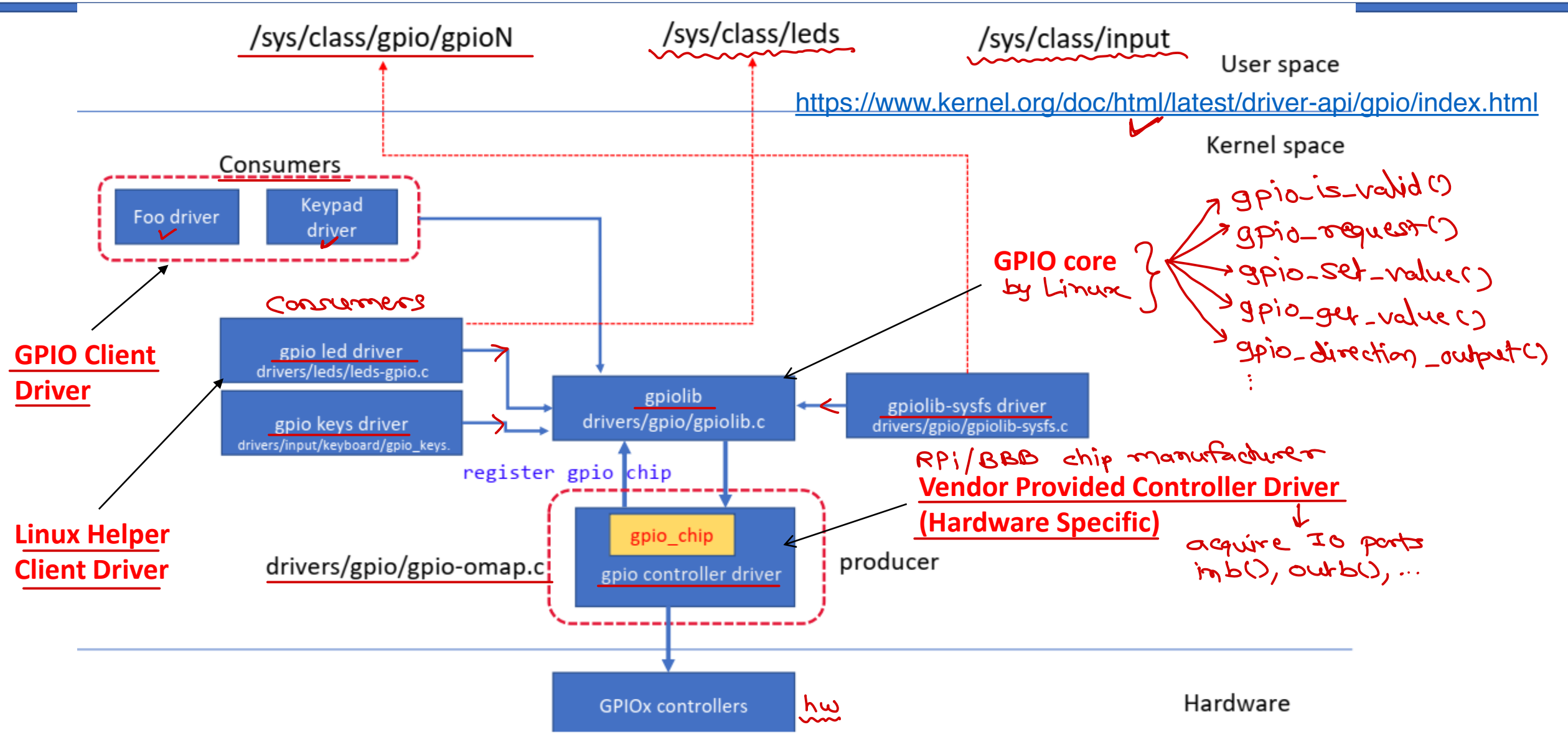
- Different buses/ Special signal for Mem & IO.
- Different instn for mem & IO access.
- e.g. x86, ..
  - IN, OUT

Hardware interaction

- IO devices are interfaced with CPU via IO ports.
  - On x86 system, this is IO mapped IO.
  - On ARM system, this is memory mapped IO.
- To ensure uniform programming, kernel provides IO access macros/functions in HAL.
- Before accessing IO ~~memory~~ port addresses, they should be owned by the driver. This can be done by *request_region()*. It can released at the end using *release_region()*.
- Actual IO operation can be done using inb(), outb(), inw(), outw(), inl(), outl(), …
- Device driver should also handle interrupts produced by the hardware device. The ISR is registered using *request_irq()*. It is released using *free_irq()*.
- ISR should not contain blocking code, because ISR runs in interrupt context. Any long running task should be deferred in tasklet, workqueue or timer (as appropriate).
- Typical hardware init and de-init code is done in open() and release() driver operation; while actual data transfer is done in read() and write() operation.

cat /proc/ioports

Linux GPIO SubSystem - producer/consumer pattern



Using Linux GPIO subsystem

- Verify the GPIO is valid or not. ***bool gpio_is_valid(int gpio_number);***
- If valid, request the GPIO from the Kernel GPIO subsystem. ***int gpio_request(unsigned gpio, const char *label);***
  - int gpio_request_one(unsigned gpio, unsigned long flags, const char *label); – Request one GPIO.
  - int gpio_request_array(struct gpio *array, size_t num); – Request multiple GPIOs.
- Export GPIO to sysfs. ***int gpio_export(unsigned int gpio, bool direction_may_change); void gpio_unexport(unsigned int gpio);***
- Set the direction of the GPIO (IN/OUT).
  - ***int gpio_direction_input(unsigned gpio);*** → switch
  - ***int gpio_direction_output(unsigned gpio, int initial_value);*** → led
- Make the GPIO to High/Low if it is set as an output pin.
  - ***gpio_set_value(unsigned int gpio, int value);***
- Set the debounce-interval and read the state if it is set as an input pin. Enable IRQ for edge/level triggered.
  - ***int gpio_get_value(unsigned gpio);***
  - ***int gpiod_set_debounce(unsigned gpio, unsigned debounce);***
  - ***int gpio_to_irq(unsigned gpio);***
  - ***request_irq() with flag IRQF_TRIGGER_RISING , IRQF_TRIGGER_FALLING, IRQF_TRIGGER_HIGH, or IRQF_TRIGGER_LOW and free_irq();***
- Release the GPIO while exiting the driver. ***void gpio_free(unsigned int gpio);***
  - void gpio_free_array(struct gpio *array, size_t num); – Release multiple GPIOs.

Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>