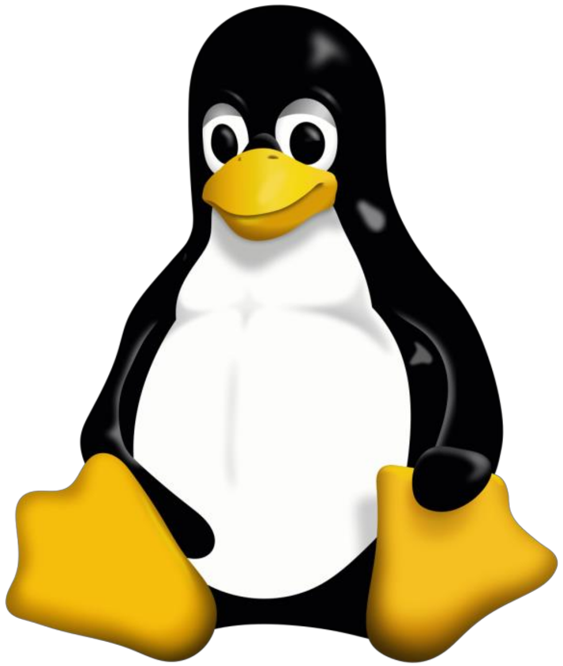


Linux Device Driver

Sunbeam Infotech

IO Ports

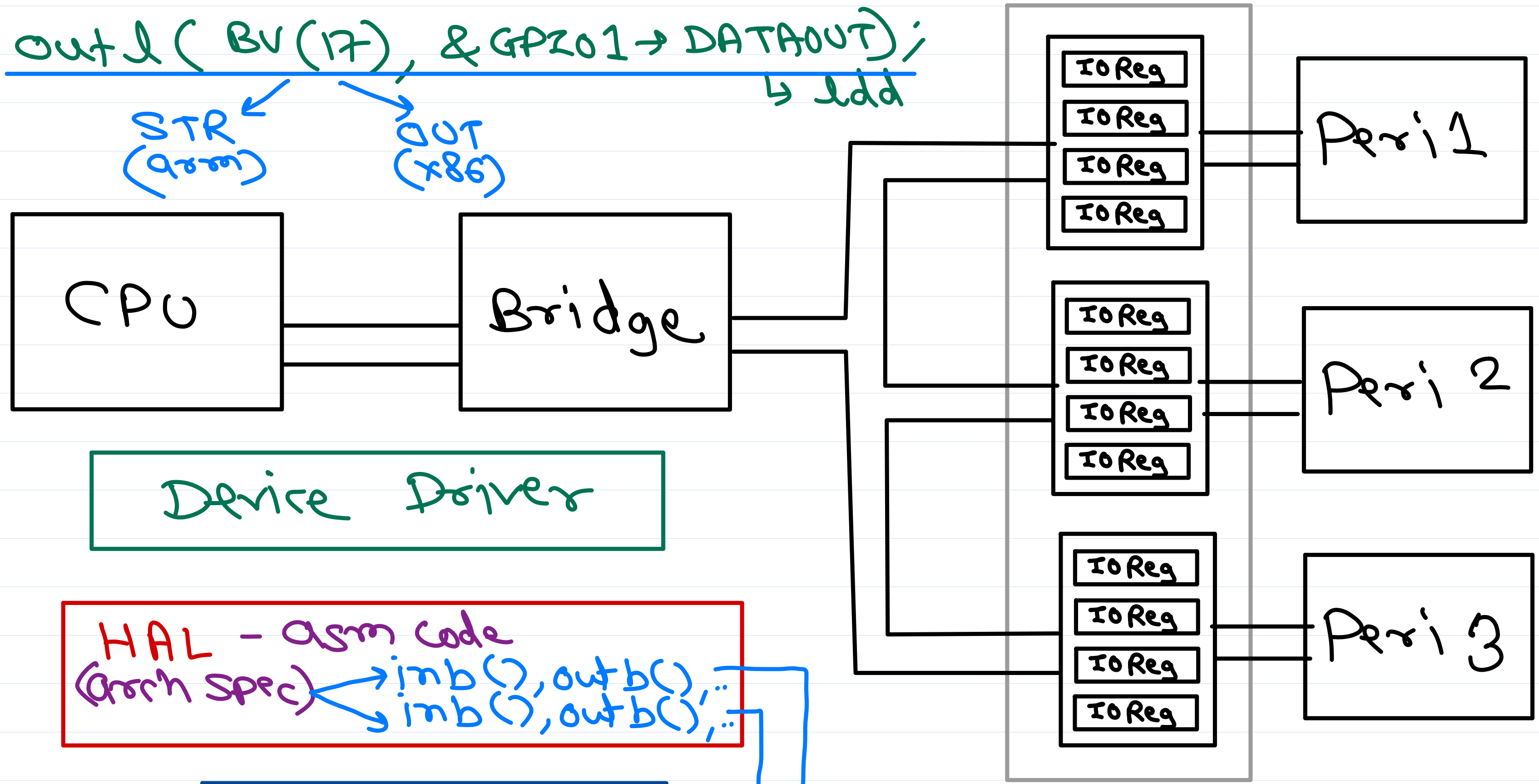
GPIO1 → DATAOUT = BV(17); → arm

outl(BV(17), &GPIO1 → DATAOUT); ↳ ldd

STR (arm)

OUT (x86)

IO Ports



Device Driver

HAL - asm code
(arch spec) → inb(), outb(), ...
→ inb(), outb(), ...

IO HW : mem mapped ← e.g. arm
OR IO mapped ← e.g. x86

Memory mapped IO

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

IO mapped IO

- Different buses/ special signal for mem & IO.
- Different instru for mem & IO access.
- e.g. x86, ..
  ↳ IN, OUT

Hardware interaction

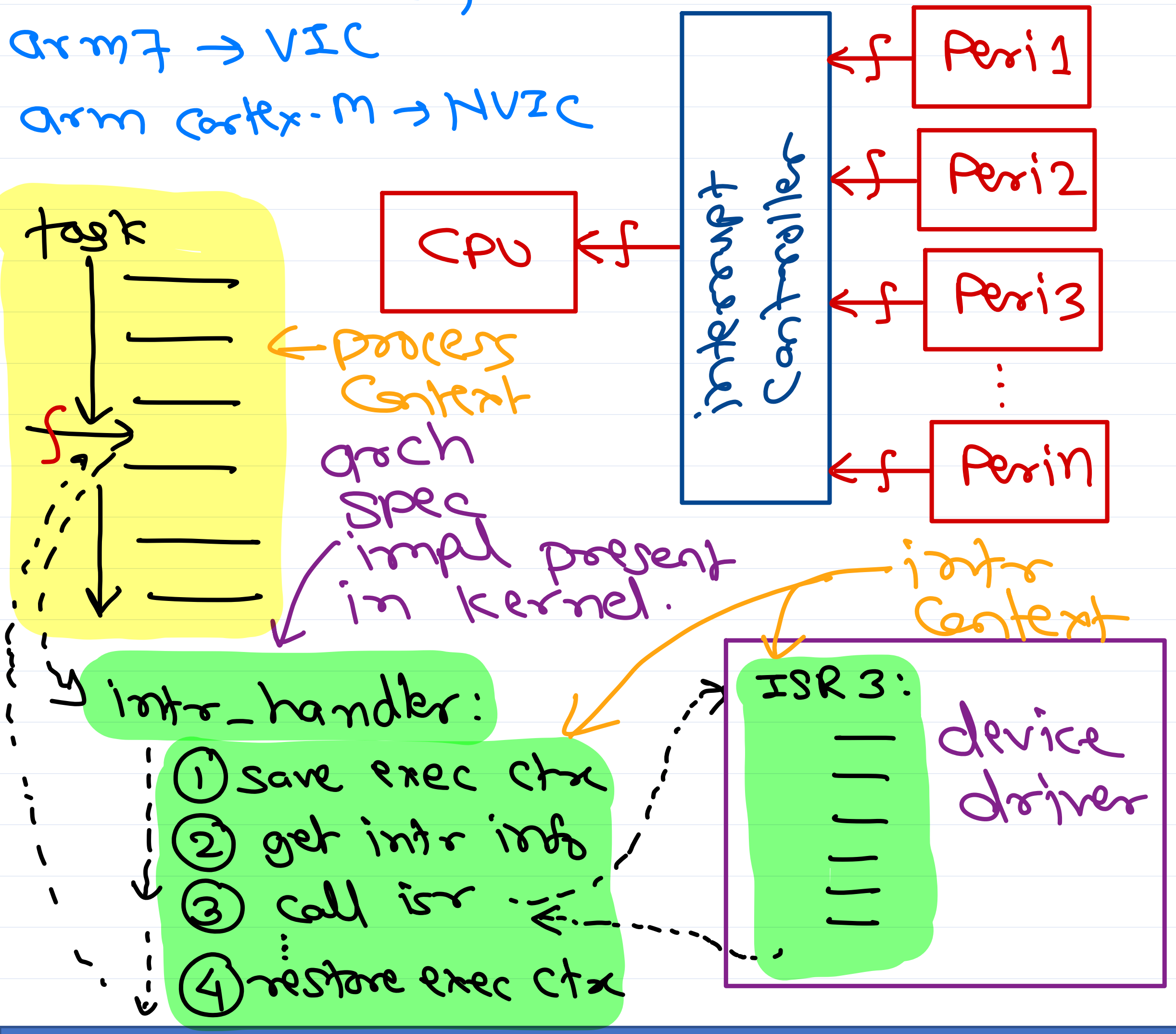
- IO devices are interfaced with CPU via IO ports.
  - On x86 system, this is IO mapped IO.
  - On ARM system, this is memory mapped IO.

cat /proc/ioports

- 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.

Interrupt Handling in Linux

x86 → PIC - 8259
arm7 → VIC
arm cortex-M → NVIC



ISR impl:

irqreturn_t isr_fn(int irq, void *param);

↳ IRQ_HANDLED → intr handled by this ISR.
↳ IRQ_NONE → intr not handled by this ISR.
↳ kernel calls next isr on same line.

Register ISR:

request_irq(irq, isr_fn, flags, name, param);

cat /proc/interrupts

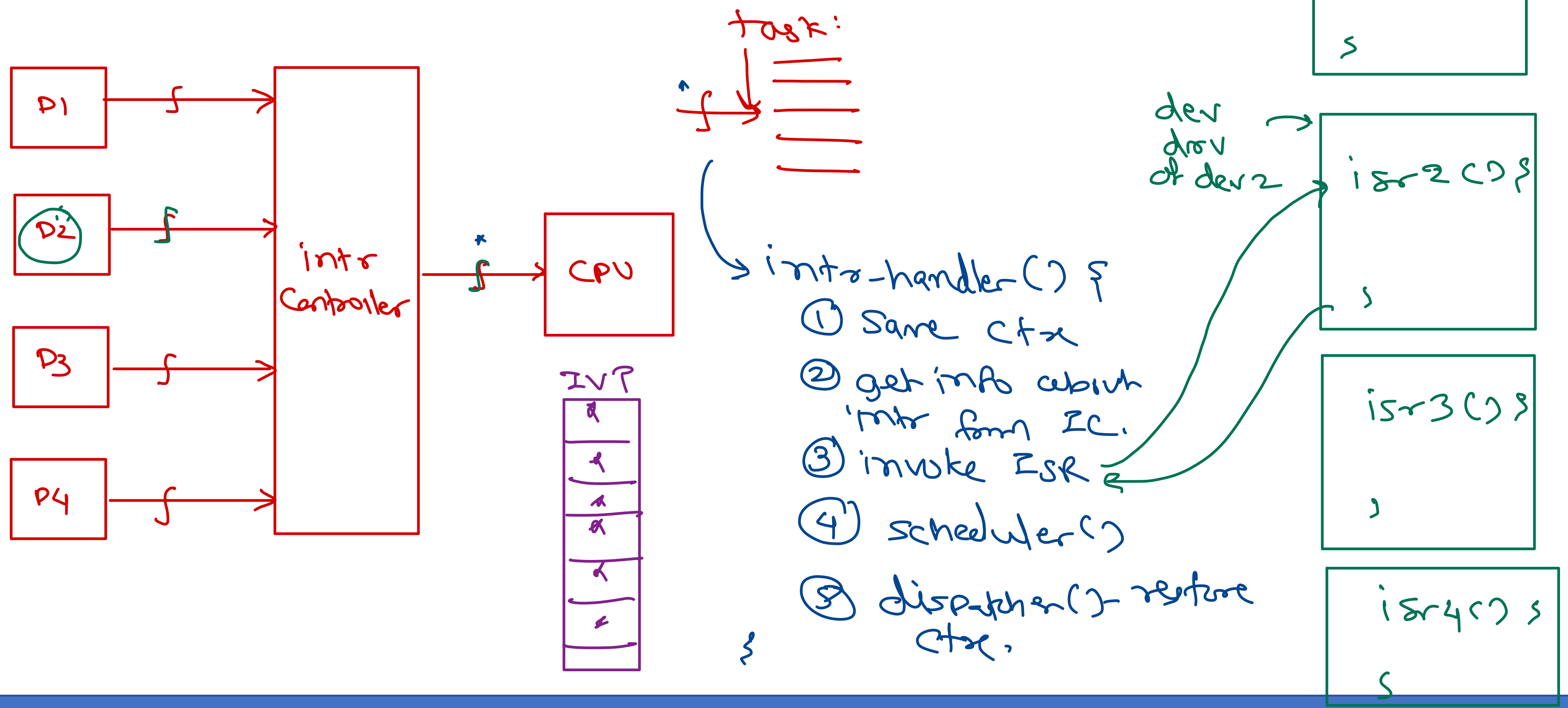
① IRQF_DISABLED
② IRQF_TIMER
③ IRQF_SAMPLE_RANDOM → random entropy pool
true random nums
↳ /dev/random
④ IRQF_SHARED
↳ same irq line is shared for multiple hw devices.
e.g. com1, com2, com3, com4 ← serial ports
4 3 4 3

Unregister ISR:

free_irq(irq, param);

Interrupt

- Interrupts are special signals sent from device to CPU.
- Interrupt handling is architecture specific.



Interrupt handling

- Interrupt is sent from the device to the PIC.
- PIC inform CPU about interrupt through interrupt line.
- CPU pause current task execution and execute interrupt handler.
- Interrupt handler does following
  - Save current task context on stack.
  - Get interrupt details from PIC.
  - Call ISR to handle the interrupt.
  - Invoke scheduler.
  - Restore the task context.
- In Linux there are two execution context.
  - Process context
    - User space process or kernel thread context. May block.
  - Interrupt context
    - Interrupt handler and ISR execution context.
    - Atomic context: cannot block.

Interrupt handling in Linux

- Since interrupt context cannot block, handler/ISR should return immediately.
- Heavy processing and/or blocking tasks should be deferred.
- Linux divides interrupt handling into two parts
  - Top half → ISR → Intr Context
    - Run immediately when interrupt arrives.
    - Do time critical and non-blocking task like interrupt acknowledgement.
    - Cannot be pre-empted by another interrupt from same device.
  - Bottom half → Soft IRQ, Tasklet, Work Queue
    - Variety of bottom half implementations in Linux kernel.
    - Execute later – in interrupt context or process context.
    - Do heavy processing and/or blocking tasks.
    - Can be pre-empted by interrupt (top-half).
- Interrupt handling must be done in corresponding device driver.
  - Driver should implement top-half and/or bottom-half as per requirement.
  - Linux kernel ensure uniform programming model irrespective of architecture.

Implementing top half

- Two step process
  - Implement ISR.
  - Register ISR.

irq_return_t my_isr(int irq, void *param) {

≡

3

request_irq()

- ISR registration
  - #include <linux/interrupt.h>
  - int request_irq(unsigned $\boxed{\text{int irq}}$, $\boxed{\text{irq\_handler\_t handler}}$, unsigned long flags, const char *name, $\boxed{\text{void *dev}}$);
    - irq: interrupt number
    - handler: typedef irqreturn_t (*irq_handler_t)(int, void *);
    - flags:
      - IRQF_DISABLED
      - IRQF_SAMPLE_RANDOM
      - IRQF_TIMER
      - IRQF_SHARED
    - name: device name /proc/irq and /proc/interrupts
    - dev: extra information to be passed to handler.
    - Returns 0 on success or –EBUSY if interrupt line is already in use.
  - request_irq() may block and should not be called from interrupt context. Typically called when opening the device for processing or module initialization.

/dev/random

multiple device instances – private obj to keep each device info

e.g. struct serial_info{
int irq;
int ioaddr;
mutex m;
...
};

struct private_struct devices[4];

irq=4, io=0x3F8, ⋮ com1	0
irq=3, io=0x2F8, ⋮ com2	1
irq=4, io=0x3E8, ⋮ com3	2
irq=3, io=0x2E8, ⋮ com4	3

com1

request_irq(4, my_isr, IRQF_SHARED, "com1", &devices[0]);

com3

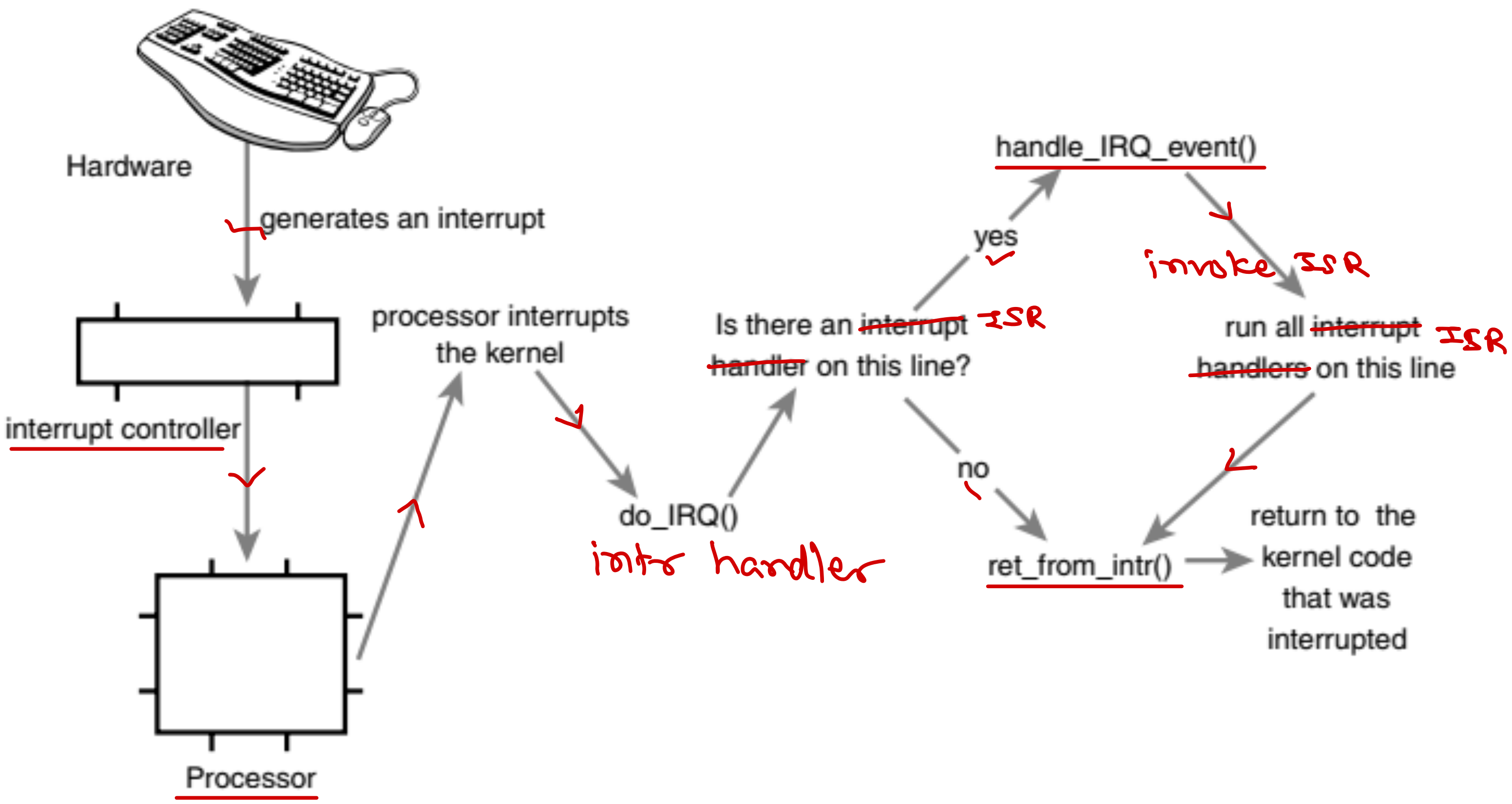
request_irq(4, my_isr, IRQF_SHARED, "com3", &devices[2]);

Implementing top half

- ISR un-registration
  - Interrupt line must be released while unloading module or closing device.
  - void free_irq(unsigned int irq, void *dev);
- Implementing ISR
  - irqreturn_t my_intr_handler(int irq, void *dev);
    - irq: interrupt number
    - dev: extra param passed while request_irq()
    - returns IRQ_HANDLED or IRQ_NONE.
  - Should contain time-critical tasks and interrupt acknowledgement.
  - Also trigger bottom-half if required.
  - Should not sleep/block.
- Linux interrupt handlers are not re-entrant. Current interrupt line is disabled while execution of ISR.
- Shared interrupt handlers
  - Must pass unique dev param – typically device private struct.
  - ISR must check if interrupt is raised from the corresponding device before handling it.
  - Kernel execute all ISR registered on same interrupt line.

Interrupt handling

- Interrupt context
  - Atomic context.
  - One page kernel stack per processor.
- Interrupt execution



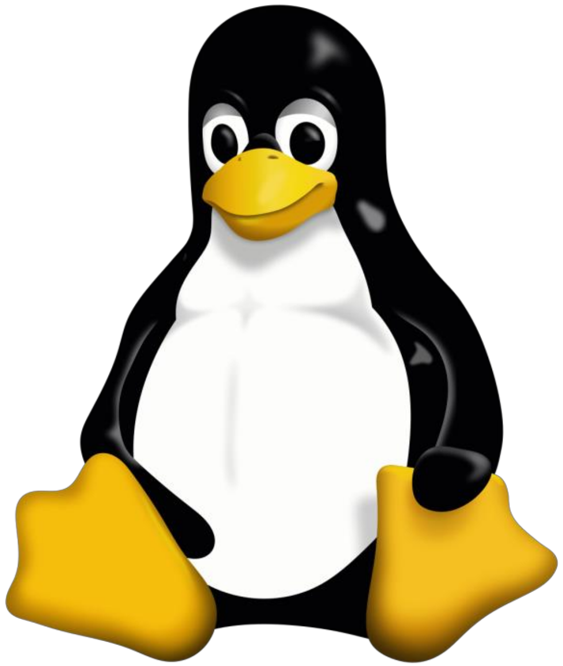
Interrupt control

- local_irq_disable(): Disables local interrupt delivery
- local_irq_enable(): Enables local interrupt delivery

current cpu

- local_irq_save(): Saves the current state of local interrupt delivery and then disables it
- local_irq_restore(): Restores local interrupt delivery to the given state
- disable_irq(): Disables the given interrupt line and ensures no handler on the line is executing
- enable_irq(): Enables the given interrupt line
- irqs_disabled(): Returns nonzero if local interrupt delivery is disabled; otherwise returns zero
- in_interrupt(): Returns nonzero if in interrupt context and zero if in process context
- in_irq(): Returns nonzero if currently executing an interrupt handler and zero otherwise

ISR

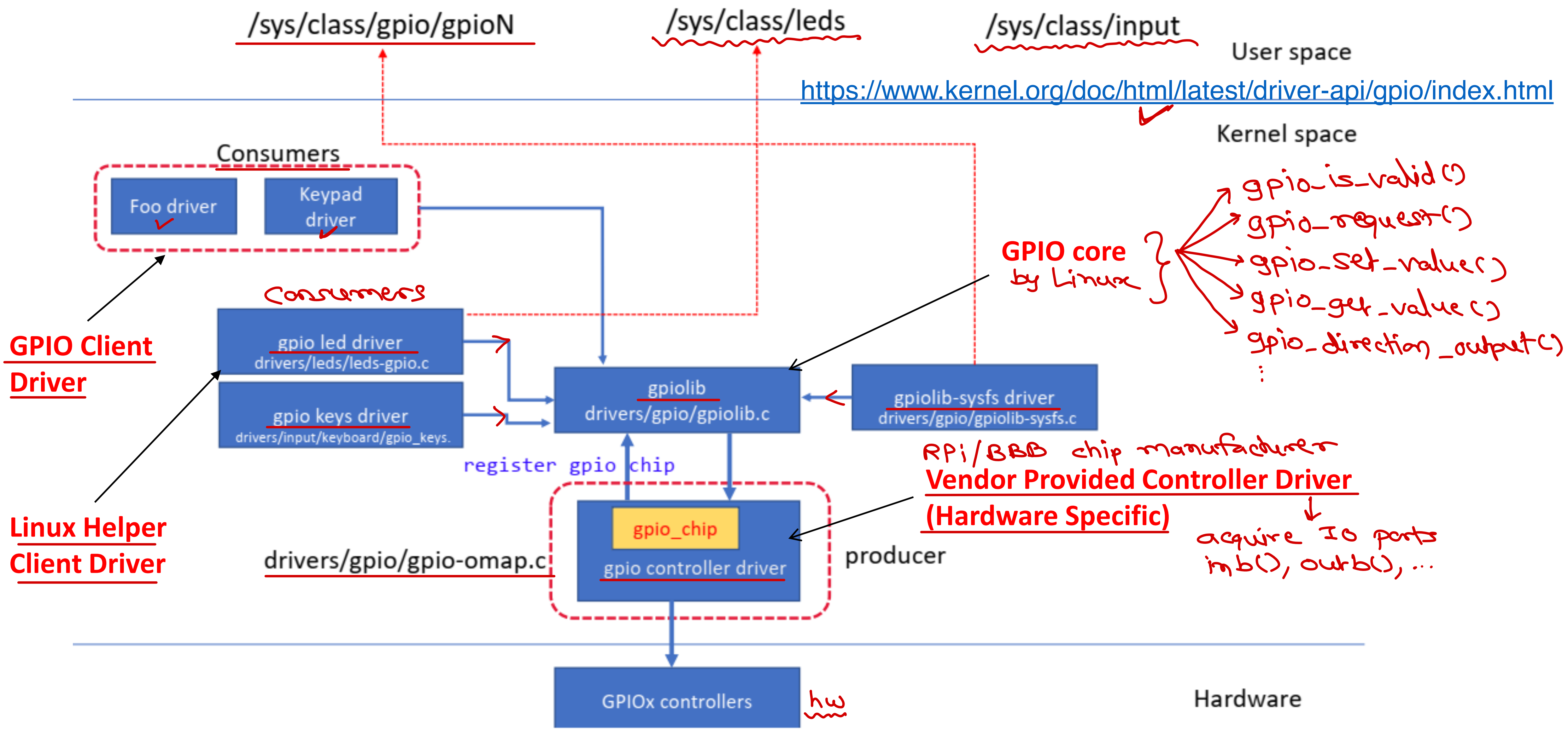


Linux GPIO SubSystem

Sunbeam Infotech

Linux GPIO SubSystem

- producer/consumer pattern



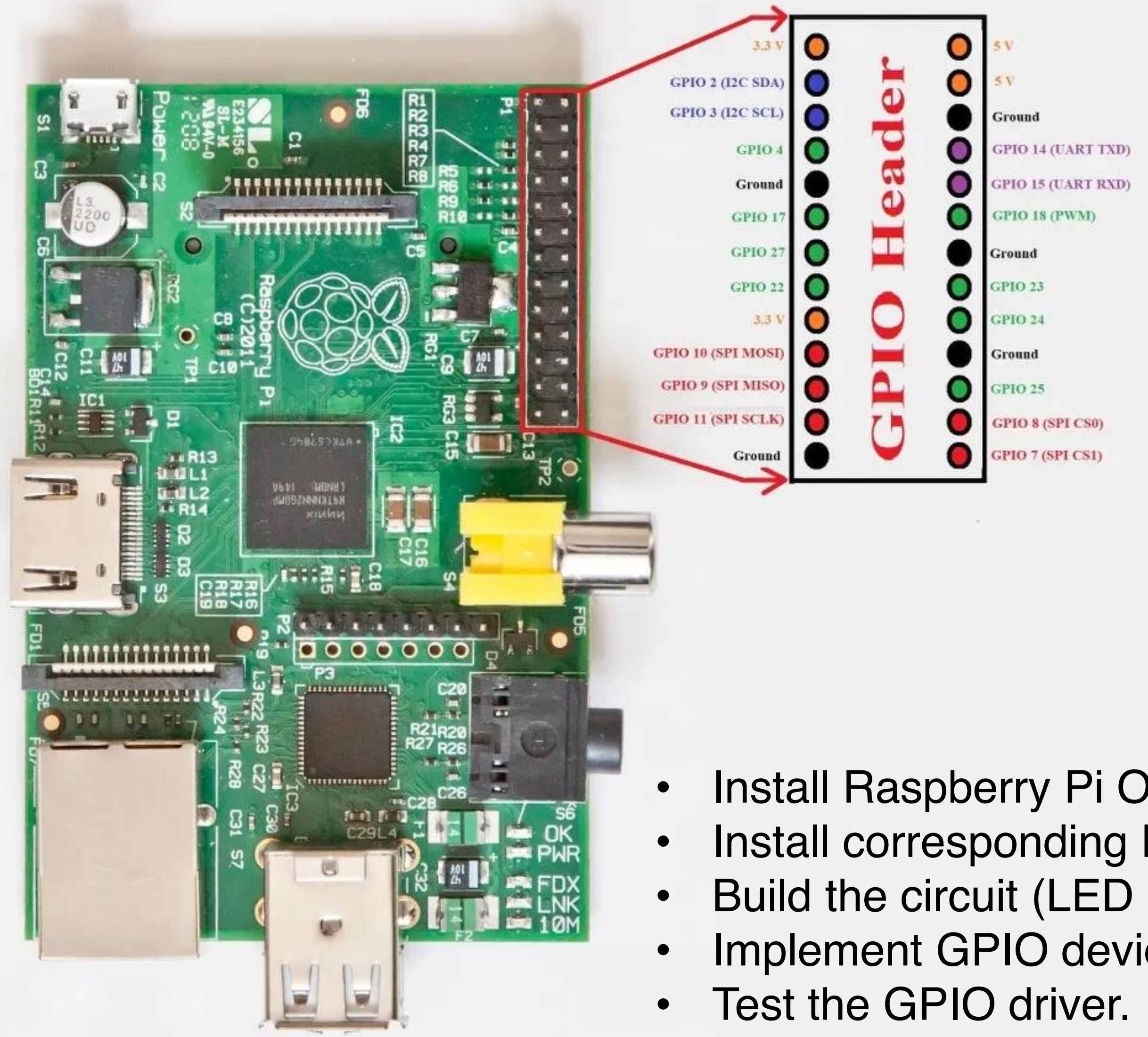
Using Linux GPIO subsystem

- 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);***

  optional
- 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.

GPIO device driver example

GPIO driver on RPi-1



- Install Raspberry Pi OS in the SD card.
- Install corresponding kernel headers (apt install raspberrypi-kernel-headers).
- Build the circuit (LED and Switch).
- Implement GPIO device driver, compile and insmod it.
- Test the GPIO driver.

BBB Led & Switch

Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>