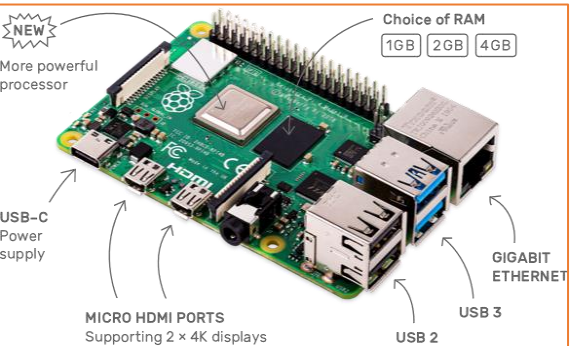
Hardware and Interrupts
Introduction to GPIO



Santosh Sam Koshy
Joint Director
C-DAC Hyderabad

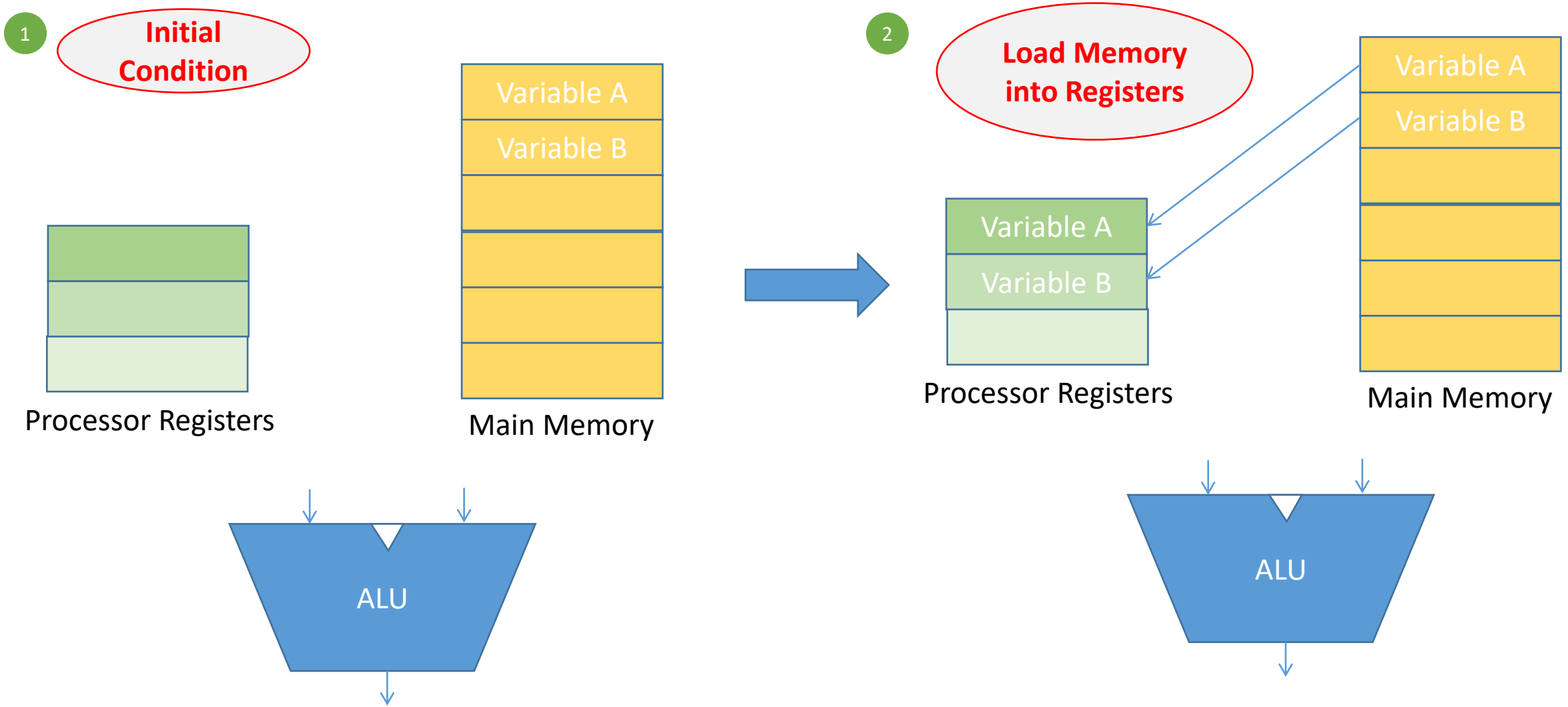
Content

- Device Description
- Memory mapped IO and IO mapped IO
- Handling Interrupts in the kernel
- BBB Header Information
- GPIO Functions
- GPIO as an Input Device
- GPIO as an Output Device

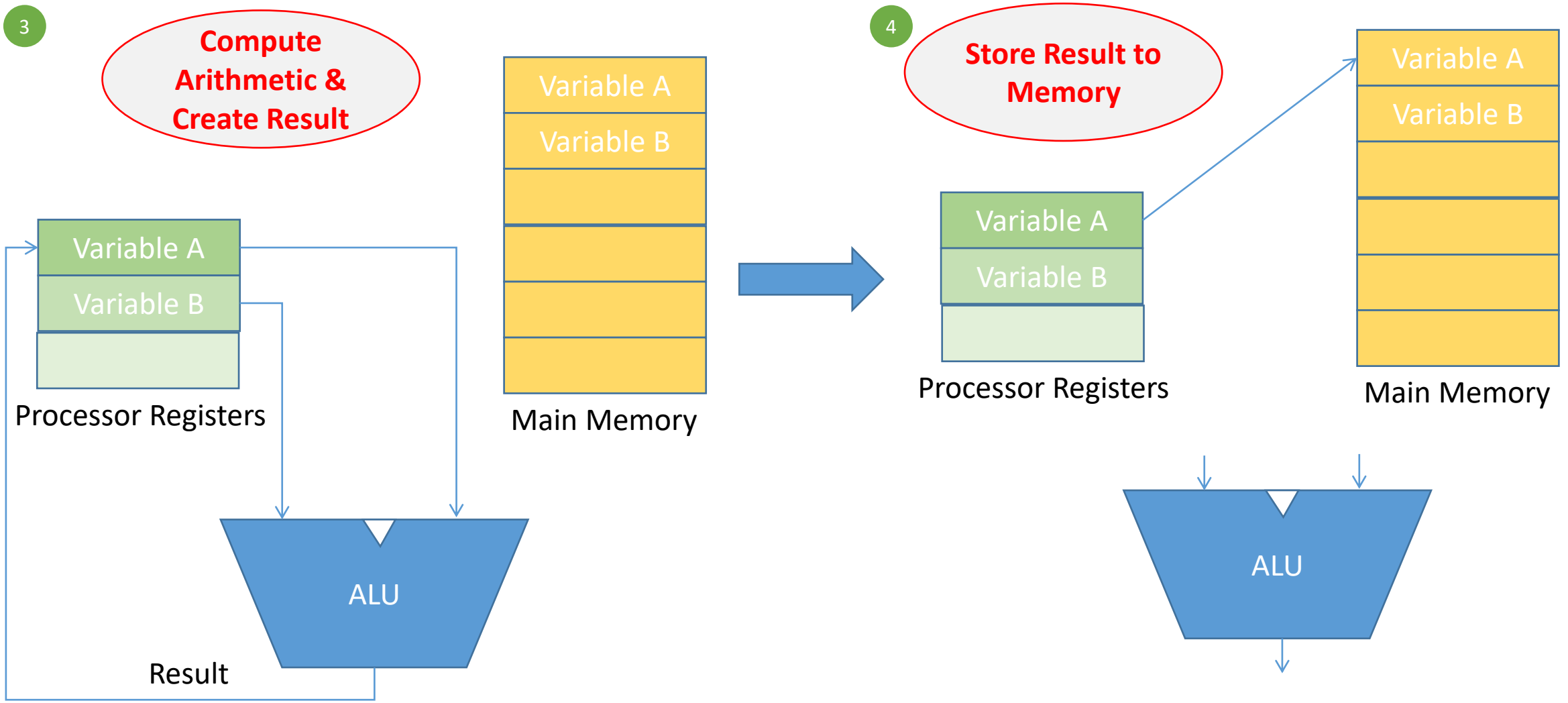
The Device Driver Abstract View

- What is a Device from the Driver's Perspective?
  - Peripheral devices are controlled by writing and reading its registers
  - Registers
    - Data Registers (Reading and Writing)
    - Control Registers (Configuring the device. Supports R&W Operations)
    - Status Registers (Maintains state of the device. Generally Read Only)
  - Every peripheral has a number of such registers
  - Therefore, a device from a driver's perspective is a set of registers that it uses to interact with, configure or transact with the device.
  - What about Registers? How are they similar/different from memory?

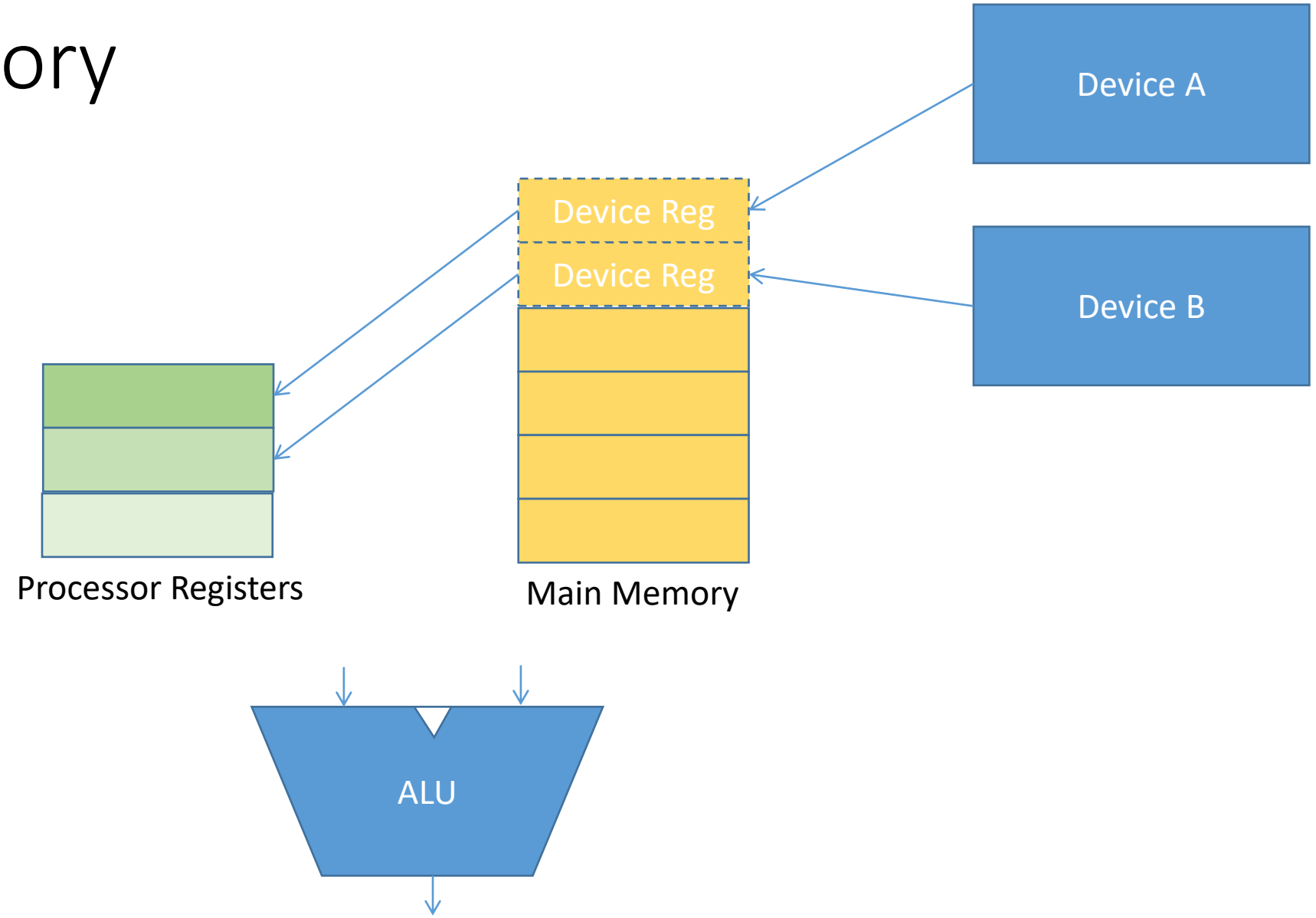
Conventional Memory



Conventional Memory



IO Memory


Device A
Device Reg
Device Reg
Device B
Processor Registers
Main Memory
ALU

Memory Caching in IO Ports

- Volatile Keyword
- Disable hardware caching during IO access. Done by the kernel itself

Data Reordering

- Consider the Following Code Snippet
  - *Write (PIN FUNCTION, CLOCK DEVICE)*
  - *Write (FREQUENCY SETTING, CLOCK DEVICE)*
  - *Write (PRESCALAR VALUE, CLOCK DEVICE)*
  - *Write (START CLOCK, CLOCK DEVICE)*
- Compiler Reorders the data for optimized performance
  - *Write (PIN FUNCTION, CLOCK DEVICE)*
  - *Write (PRESCALAR VALUE, CLOCK DEVICE)*
  - *Write (START CLOCK, CLOCK DEVICE)*
  - *Write (FREQUENCY SETTING, CLOCK DEVICE)*

Barriers

- The code will be modified like this
  - *Write (PIN FUNCTION, CLOCK DEVICE)*
  - *Write (FREQUENCY SETTING, CLOCK DEVICE)*
  - *Write (PRESCALAR VALUE, CLOCK DEVICE)*
  - *BARRIER();*
  - *Write (START CLOCK, CLOCK DEVICE)*

Barriers in Linux Kernel

- The linux kernel provides 1 macro for compiler re-ordering
  - void barrier(void)
    - The kernel makes it a point that the compiler optimizations are absent across the barrier. The memory values present in the CPU registers are immediately stored in the memory. These do not prevent hardware barriers and therefore, the hardware is free to re-order the code
  - Header - #include <linux/kernel.h>
- 4 macros are provided by the kernel for hardware memory re-ordering
  - void rmb(void);
  - void read_barrier_depends(void);
  - void wmb(void);
  - void mb(void);
    - These functions insert hardware memory barriers in the compiled instruction flow. They are supersets of the barrier macro.
    - An rmb guarantees that any reads appearing before the barrier are completed prior to the execution of any subsequent read.
    - The _depends read memory barrier inserts a barrier only if the read is dependent on data from other reads. This is platform specific and so, be careful on using it
    - wmb orders write and mb orders both read and write.
  - Header - #include <asm/system.h>

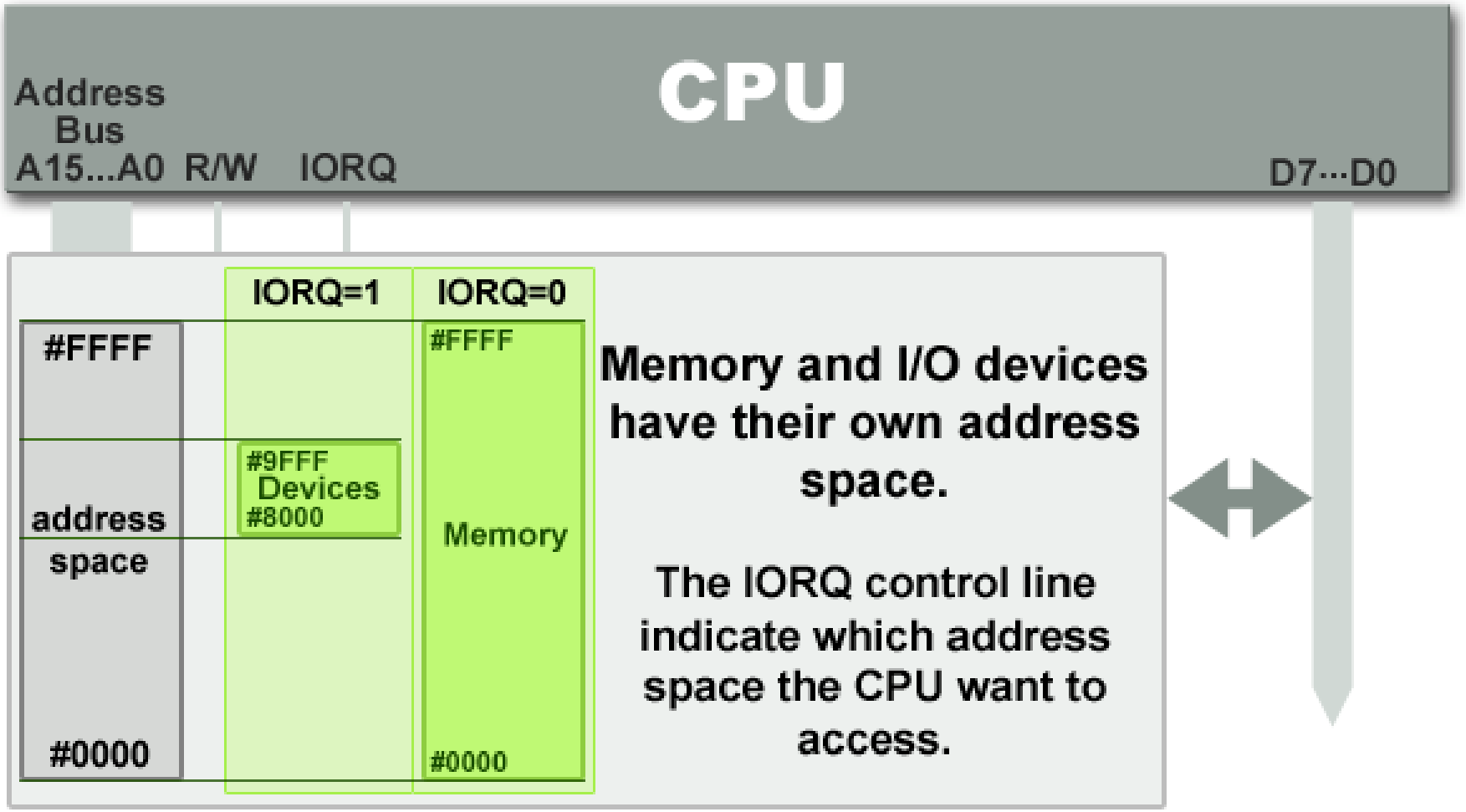
The Device Driver Abstract View

- How to Accesses the Registers of a Device?
  - These are generally mapped contiguously in the Processor's Memory Space or IO Address Space, based on the processor architecture or device architecture
    - Processor architecture – x86 family supports both I/O Ports and Memory mapped IO, ARM supports only Memory Mapped IO
    - Device Architecture – PCI devices are always memory mapped
  - Drivers have to gain access to the device memory. For this,
    - Drivers need to know the mapping – Whether it is I/O Mapped Device or Memory Mapped Device
    - Drivers need to know the starting address of the mapping
    - Drivers need to know
      - the size of the registers,
      - offsets of specific registers and
      - how to access and transact with them – read/write

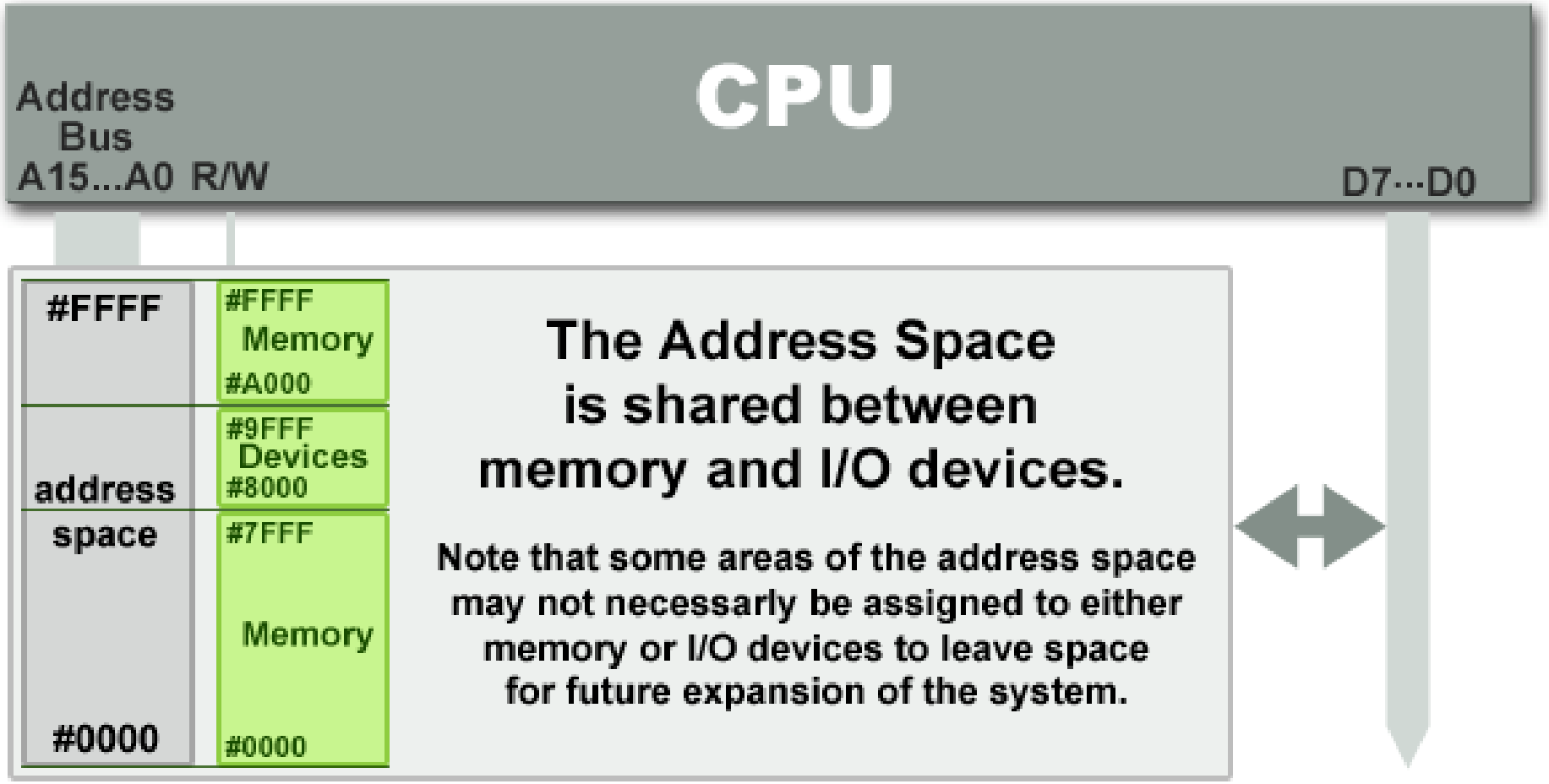
Device Addressing

- IO Mapped IO or IO Ports
  - Peripheral devices are mapped into IO space and are accesssed through special instructions
  - Additional control lines are required to segregate IO access
  - Memory is better utilized
- Memory Mapped IO or IO Memory
  - Peripheral devices are mapped into the system memory
  - All access to this device space is performed using similar instructions that access memory
  - Effective memory gets reduced

IO mapped IO (IOIO) or IO Ports



Memory Mapped IO (MMIO) or IO Memory



Communicating with IO Ports

- Drivers communicate with many devices through the I/O ports.
- Linux introduces a set of functions that may be used to gain exclusive access to an I/O region and communicate data transfers to and from these ports.
- Before embarking into communication with the I/O ports, the driver must gain access to the required ports by calling the function
  - struct resource *request_region(unsigned long first, unsigned long n, const char *name);
    - *The argument first is the first address requested*
    - *The argument long specifies the length of the addresses requested*
    - *The argument name identifies the requested region by this name in /proc/ioports*
- Header - #include <linux/ioport.h>

Communicating with IO Ports

- When the I/O ports have been used as per requirement, they should be returned to the kernel so that other drivers may avail their existence. The I/O ports are returned by the function
  - void release_region(unsigned long start, unsigned long n);
- Before using a I/O region, a check on the availability has to be conducted to be certain that the requested region will be available for our use. A special function allows this check.
  - int check_region(unsigned long first, unsigned long n);
    - *This function returns a negative error code if the region is not available. This is a deprecated function and may not always prove to be true since it does not run atomic with the request_region function.*

Communicating with IO Ports

- On successfully being allotted the region, the actual communication to the ports may be carried out using the kernel provided functions.
- These functions are specific to the port sizes on the I/O devices and provide interfaces for 8-bit, 16-bit and 32-bit ports
  - unsigned inb (unsigned port);
  - unsigned inw (unsigned port);
  - unsigned inl (unsigned port);
    - *Get data from the port specified as an argument*
  - void outb (unsigned char byte, unsigned port);
  - void outw (unsigned short word, unsigned port);
  - void outl (unsigned long word, unsigned port);
    - *Send data to the port specified in the argument*
- Header - #include <asm/io.h>

Communicating with IO Ports

- The kernel also supports string operations that render its services in allowing a string of ‘n’ bytes to be transferred between the driver and the I/O port
  - void insb (unsigned port, void *addr, unsigned long count);
  - Similarly..insw, insl
    - *Copies count bytes of information from the port to addr*
  - void outsb(unsigned port, void *addr, unsigned long count);
    - *Write count data pointed by addr to the port*
- The kernel provides mechanisms of synchronization between a high-end processor and a relatively slower I/O by allowing a pause functionality in data transfer.
- This feature may be accessed by ending the function names previously discussed with an ‘_p’, such as inb_p, outb_p....

IO Memory

- IO Memory regions must be allocated prior to use
  - struct resource *request_mem_region(unsigned long start, unsigned long len, char *name);
- IO Memory regions should when no longer in use
  - void release_mem_region(unsigned long start, unsigned long len);
- Accessing IO Memory regions should be preceded by a call to
  - void *ioremap(unsigned long phys_addr, unsigned long size);
  - void *ioremap_nocache(unsigned long phys_addr, unsigned long size);
- After accessing the IO Memory region, unmap it using
  - void iounmap(void * addr);

IO Memory

- To read from IO memory,
  - unsigned int ioread8(void *addr);
  - unsigned int ioread16(void *addr);
  - unsigned int ioread32(void *addr);
- Writing to the IO memory through
  - void iowrite8(u8 value, void *addr);
  - void iowrite16(u16 value, void *addr);
  - void iowrite32(u32 value, void *addr);
- Reading a series of values
  - void ioread8_rep(void *addr, void *buf, unsigned long count);
  - void iowrite8_rep(void *addr, const void *buf, unsigned long count);

The Device Driver Abstract View

- How do drivers deal with speed differences between the processor and the device?
  - Processor may be too fast, device may be too slow to respond. Transmission may slow down the processor.
  - Device generates data asynchronously, processor does not know when the data is ready.
  - Solution: Interrupts – Mechanisms for the device to intimate to the processor that it needs attention.

Interrupts - Definition

- Means to notify the kernel that a device is requesting attention.
- Characteristics of Interrupts
  - Interrupts occur Asynchronously
    - Processor completes the current instruction being executed
    - Jumps to Interrupt Service Routine
    - Handle the Interrupt
    - Return from Interrupt
  - Global Interrupts are disabled. It may be enabled in the handler by the driver
  - Interrupts must be handled quickly because the longer interrupts take to execute, the longer interrupts remain disabled
  - No Sleep, or other delay functions should be called in an interrupt
  - Less important functions of the interrupt should be performed in a bottom half (tasklet)

Interrupts

- Components
  - IRQ Line – Interrupt Number
    - These are specific to a peripheral
    - It could be shared between number of devices
    - Limited number (32 in x86). You have to find out your IRQ number before you can use it. It is very specific to the hardware that you are using
  - Interrupt Handler
    - Each handler has to register itself with the kernel whenever an operation is to be performed on the device.
    - Registering a handler is a notification by the driver to the kernel, claiming authority for access to the device through a requested IRQ number.
    - On completion of access to the device, the handler may be unregistered and the irq number freed for allowing access to other applications.

Interrupts – Requesting an IRQ and binding a handler

- Registering an interrupt handler to an irq number and notifying the kernel is done by
  - int request_irq (unsigned int irq,
    irqreturn_t (*handler) (int, void *, struct pt_regs *),
    unsigned long flags,
    const char *dev_name,
    void *dev_id);
    - The value returned from request_irq to the is either 0 to indicate success or a negative error code, as usual.
    - It is not uncommon for the function to return -EBUSY to signal that another driver is already using the requested interrupt line.
- The handler is freed by calling
  - void free_irq (unsigned int irq, void *dev_id);

Interrupts – FLAGS passed to request_irq

- FLAGS
  - SA_INTERRUPT (May be deprecated.. Just check)
    - This bit indicates a “fast” handler.
    - Fast interrupt handlers run with all interrupts disabled on the local processor
  - SA_SHIRQ
    - This bit indicates that the interrupt can be shared between devices.
    - The dev_id must be unique to each registered handler. A pointer to any per-device structure is sufficient. NULL cannot be passed to this field
    - The interrupt handler must be capable of detecting whether its device generated an interrupt. This requires both hardware support and associated logic in the handler
  - SA_SAMPLE_RANDOM
    - Adds to the kernel entropy pool to increase the randomness.

Interrupts – The Handler Function

- The Interrupt Handler
  - static irqreturn_t intr_handler (int irq, void *dev_id, struct pt_regs *regs);
    - *The return value of an interrupt handler is the special type irqreturn_t.*
    - *An interrupt handler can return two special values, IRQ_NONE or IRQ_HANDLED.*
    - *IRQ_NONE is returned when the handler detects an interrupt for which its device was not the originator.*
    - *IRQ_HANDLED is returned if the interrupt handler was correctly invoked*

Interrupts – Where to request for IRQ

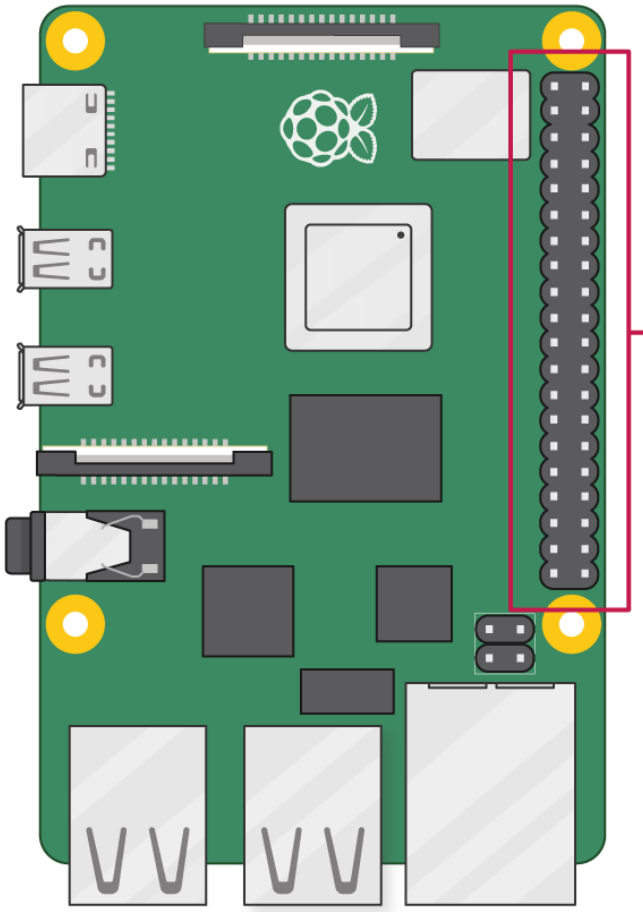
- There are two places where an IRQ can be requested.
  - During the driver initialization at module_init function
    - For this, it is advisable that the interrupt is requested as a shared IRQ
  - In the open method of the driver
    - If called in the open call, it allows the use of the interrupt line only when the application requests for it.
    - The disadvantage of this technique is that a per device open count has to be maintained to know when interrupts can be disabled.

Interrupts – Disabling and Re-enabling

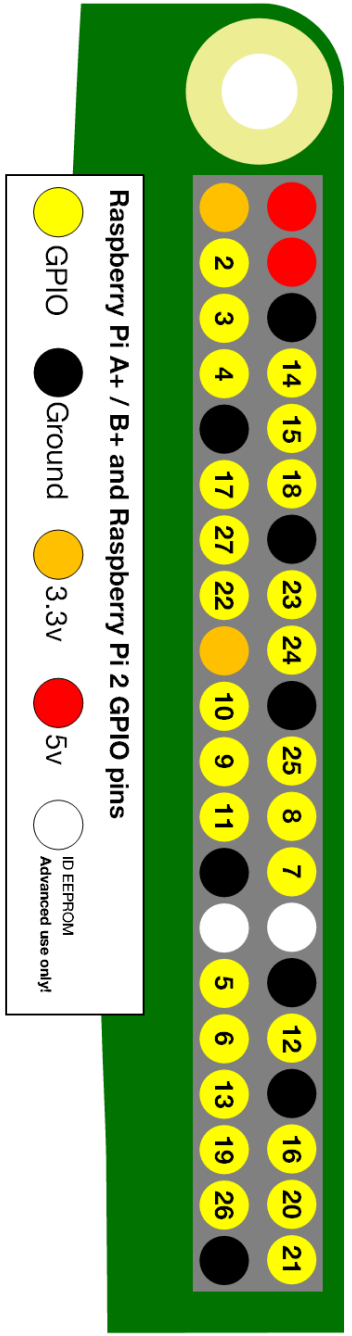
- Single Interrupts
  - Sometimes the driver may need to disable interrupt delivery for a specific line. This is achieved by using one of the functions
    - void disable_irq(int irq);
    - void enable_irq(int irq);
- Disable all Interrupts
  - void local_irq_save(unsigned long flags);
  - void local irq_disable(void);
    - shuts off the interrupts
  - void local_irq_restore(unsigned long flags);
  - void local_irq_enable(void);
    - Re-enables the interrupts

Doing it on the Raspberry Pi4

GPIO Available



3V3 power	1	2	5V power
GPIO 2 (SDA)	3	4	5V power
GPIO 3 (SCL)	5	6	Ground
GPIO 4 (GPCLK0)	7	8	GPIO 14 (TXD)
Ground	9	10	GPIO 15 (RXD)
GPIO 17	11	12	GPIO 18 (PCM_CLK)
GPIO 27	13	14	Ground
GPIO 22	15	16	GPIO 23
3V3 power	17	18	GPIO 24
GPIO 10 (MOSI)	19	20	Ground
GPIO 9 (MISO)	21	22	GPIO 25
GPIO 11 (SCLK)	23	24	GPIO 8 (CE0)
Ground	25	26	GPIO 7 (CE1)
GPIO 0 (ID_SD)	27	28	GPIO 1 (ID_SC)
GPIO 5	29	30	Ground
GPIO 6	31	32	GPIO 12 (PWM0)
GPIO 13 (PWM1)	33	34	Ground
GPIO 19 (PCM_FS)	35	36	GPIO 16
GPIO 26	37	38	GPIO 20 (PCM_DIN)
Ground	39	40	GPIO 21 (PCM_DOUT)



Introducing GPIO on RPi4

- Header - #include <linux/gpio.h>
- Functions
  - static inline bool gpio_is_valid(int number)
    - check validity of GPIO number
  - static inline int gpio_request(unsigned gpio, const char *label)
    - allocate the GPIO number, the label is for sysfs
  - static inline void gpio_free(unsigned gpio)
    - deallocate the GPIO line
  - static inline int gpio_export(unsigned gpio, bool direction_may_change)
    - make available via sysfs and decide if it can change from input to output and vice versa
  - static inline void gpio_unexport(unsigned gpio)
    - remove from sysfs

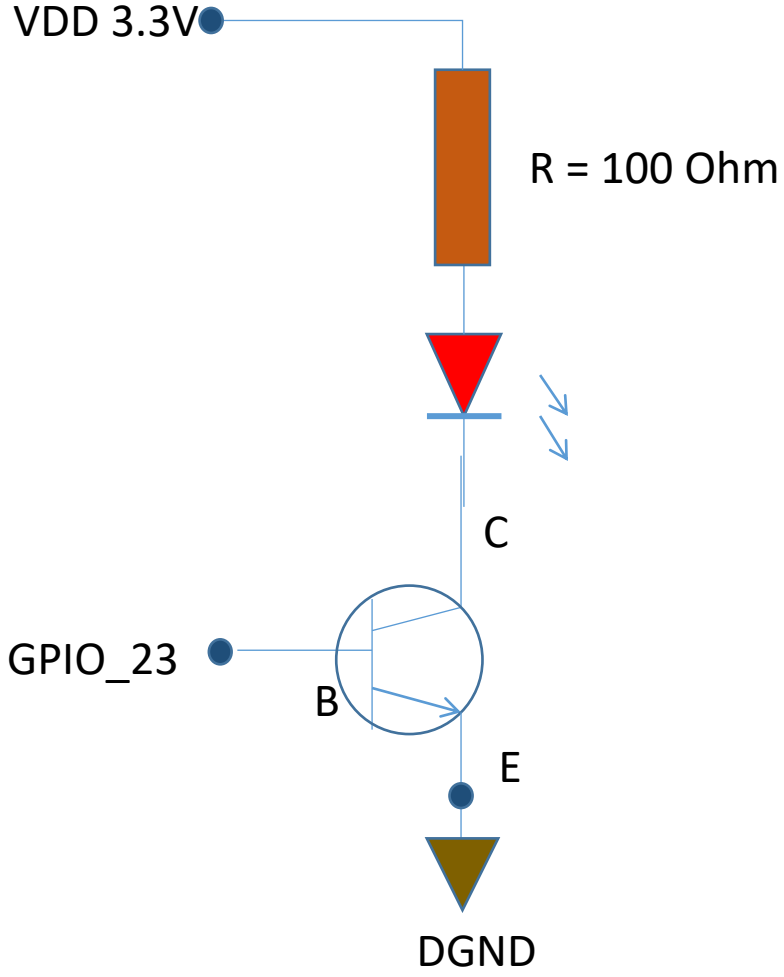
Introducing GPIO on RPi4

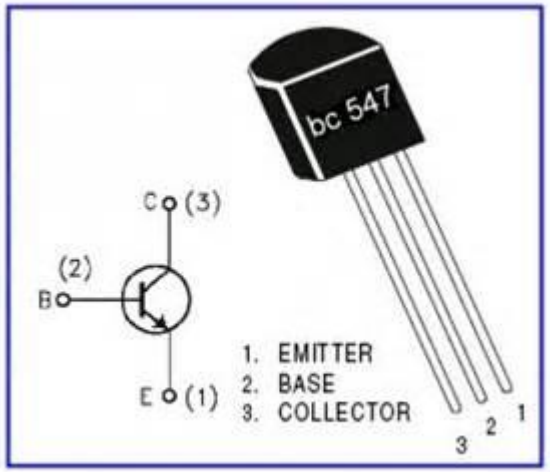
- static inline int gpio_direction_input(unsigned gpio)
  - an input line (as usual, return of 0 is success)
- static inline int gpio_get_value(unsigned gpio)
  - get the value of the GPIO line
- static inline int gpio_direction_output(unsigned gpio, int value)
  - value is the state
- static void gpio_set_value(unsigned gpio, value)
  - Set the value of the GPIO line
- static inline int gpio_set_debounce(unsigned gpio, unsigned debounce)
  - set debounce time in ms (platform dependent)
- static inline int gpio_sysfs_set_active_low(unsigned gpio, int value)
  - set active low (invert operation states)
- static inline int gpio_to_irq(unsigned gpio)
  - associate with an IRQ

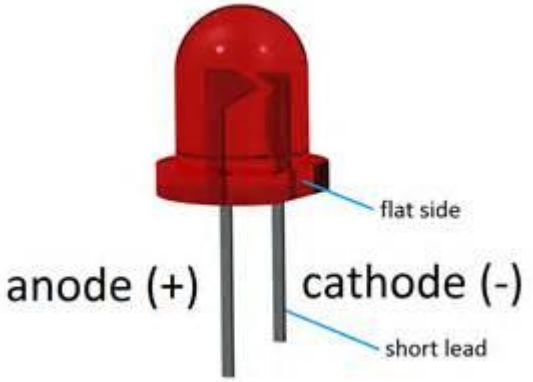
Case Study – 1: Variable Frequency LED Toggle

- Involves IOCTL, Kernel Timers and GPIO
- Toggle an LED connected to GPIO_23 of your RPi4 in a specific periodicity
- Use Kernel Timers to maintain toggle periodicity
- Using IOCTL from the user space, change the frequency of toggling by sending out a command through the driver to the LED
- Demonstrate…

Required Circuit







Case Study – 2: Toggle an LED when Switch is Pressed

- Connect a tactile switch to GPIO_24 on your RPi4
- The switch should be pulled up, through a 15K Ohm resistor
- Whenever the switch is pressed, an interrupt should be generated and in the handler, the LED should be toggled
- Register the corresponding interrupt handler and IRQ number through GPIO access

Required Circuit