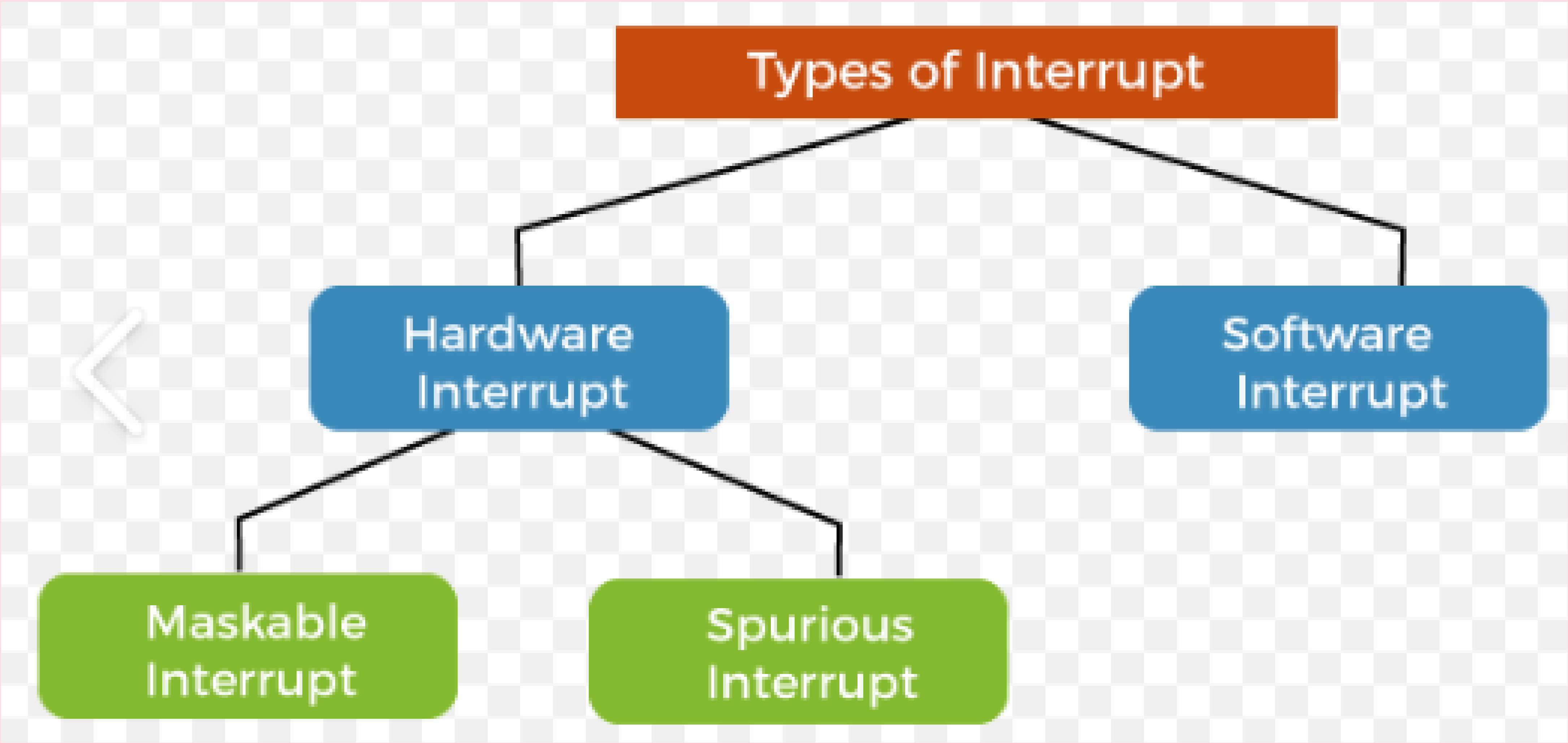
INTERRUPT MANAGEMENT

Efficient Interrupt Management: Fundamentals and Techniques

INTRODUCTION

An interrupt is a signal from hardware or software that prompts the processor to pause its current task and handle a higher-priority process. The processor saves the current task's address, loads the address of the Interrupt Service Routine (ISR), and processes the interrupt before resuming the interrupted task. The delay in handling the interrupt is called Interrupt Latency, caused by saving registers and notifying the device to stop sending the interrupt signal.

Types of Interrupts



Types of Interrupts

Software Interrupts:

1. Initiated by software or the system, these interrupts signal the operating system to perform a task or handle an error. They occur due to specific instructions or exceptions (e.g., division by zero). A special "interrupt instruction" is used, causing the processor to halt its work and switch to an interrupt handler, which completes the task and returns control.

Hardware Interrupt:

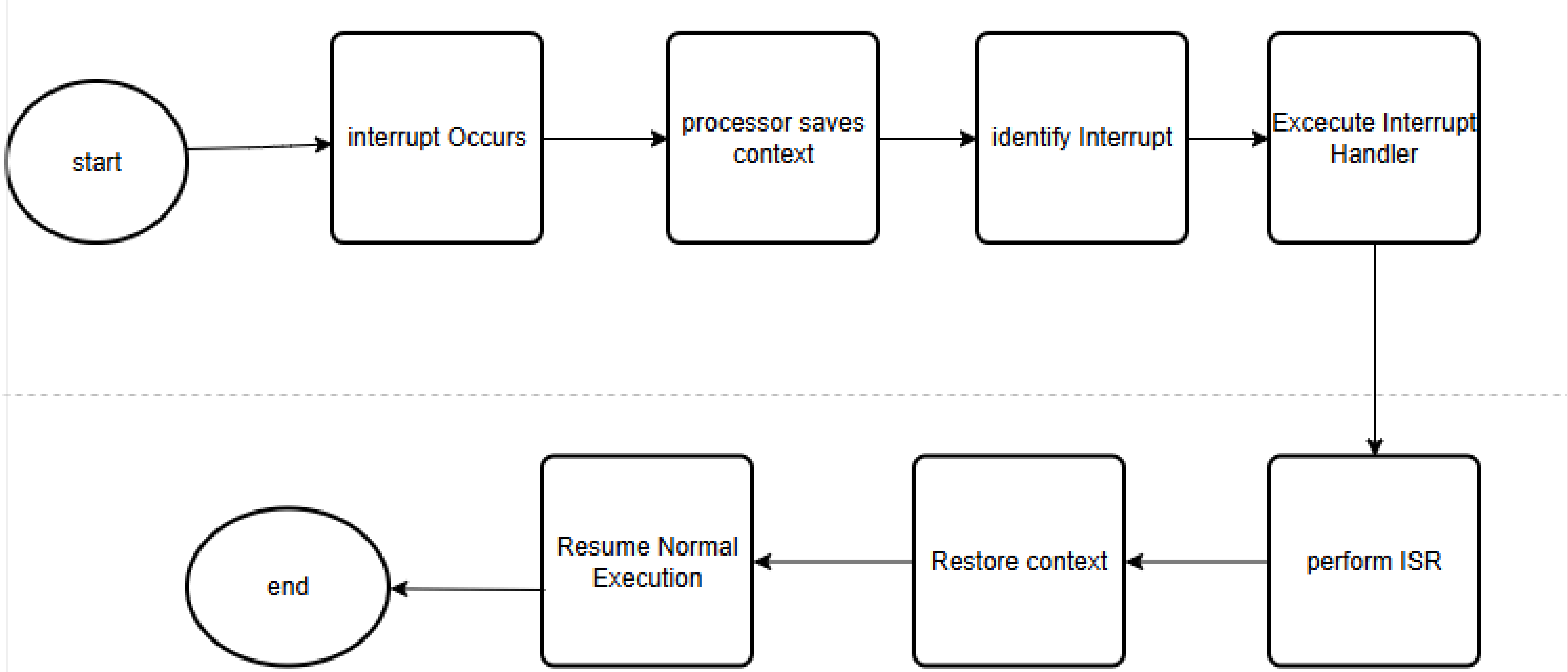
2. Triggered by devices connected to an Interrupt Request Line, hardware interrupts can be of two types:
   - Maskable Interrupt: Can be enabled or disabled using an interrupt mask register.
   - Spurious Interrupt: Occurs without a clear source, often due to wiring issues or malfunctioning systems.

IMPORTANCE OF INTERRUPTS

- Efficiency: Avoids continuous polling.
- Real-time response: Handles critical events quickly.

FLOWCHART OF INTERRUPT HANDLING MECHANISM

The Image below depicts the flowchart of interrupt handling mechanism

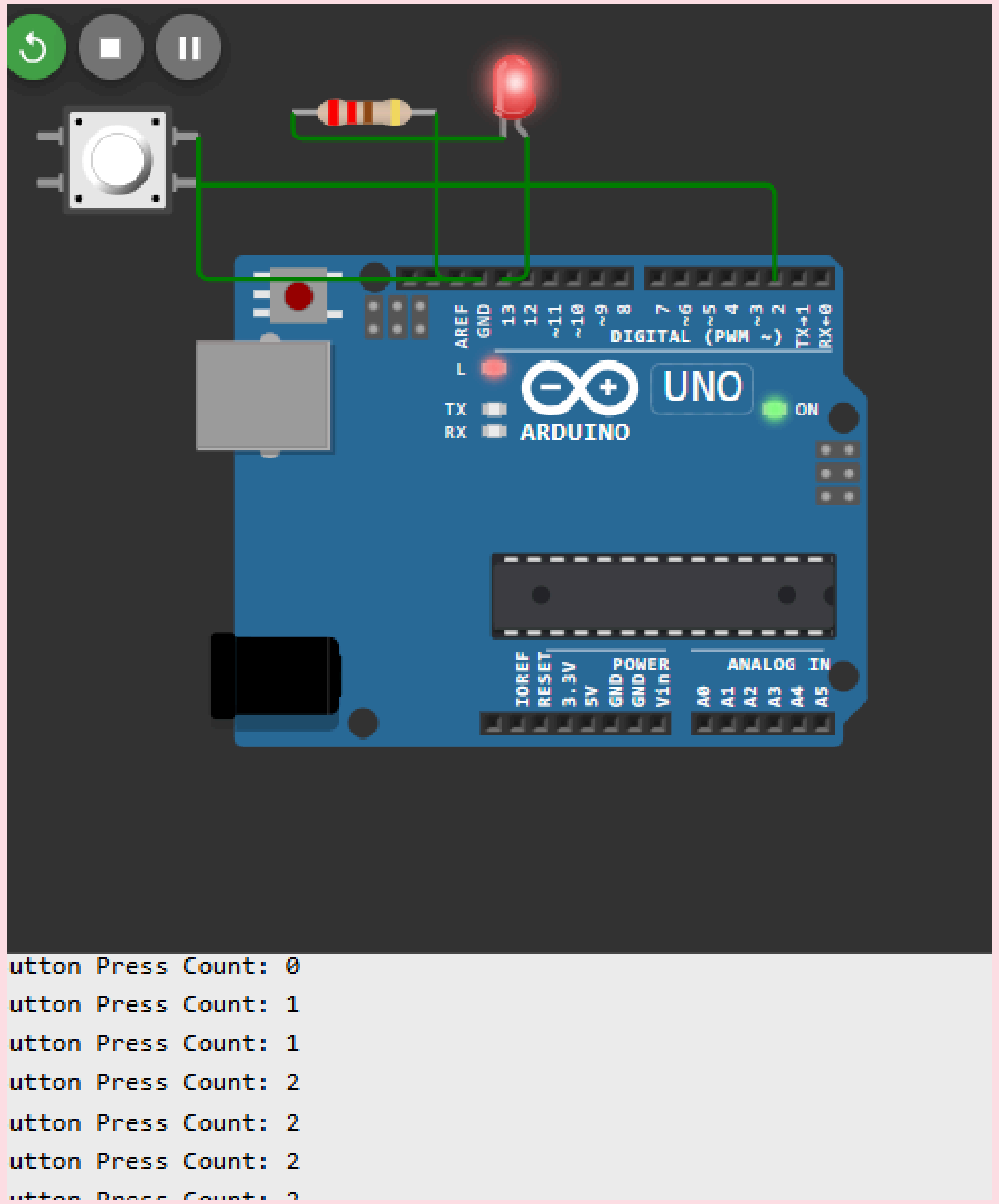


Step 1:- Any time that an interrupt is raised, it may either be an I/O interrupt or a system interrupt.

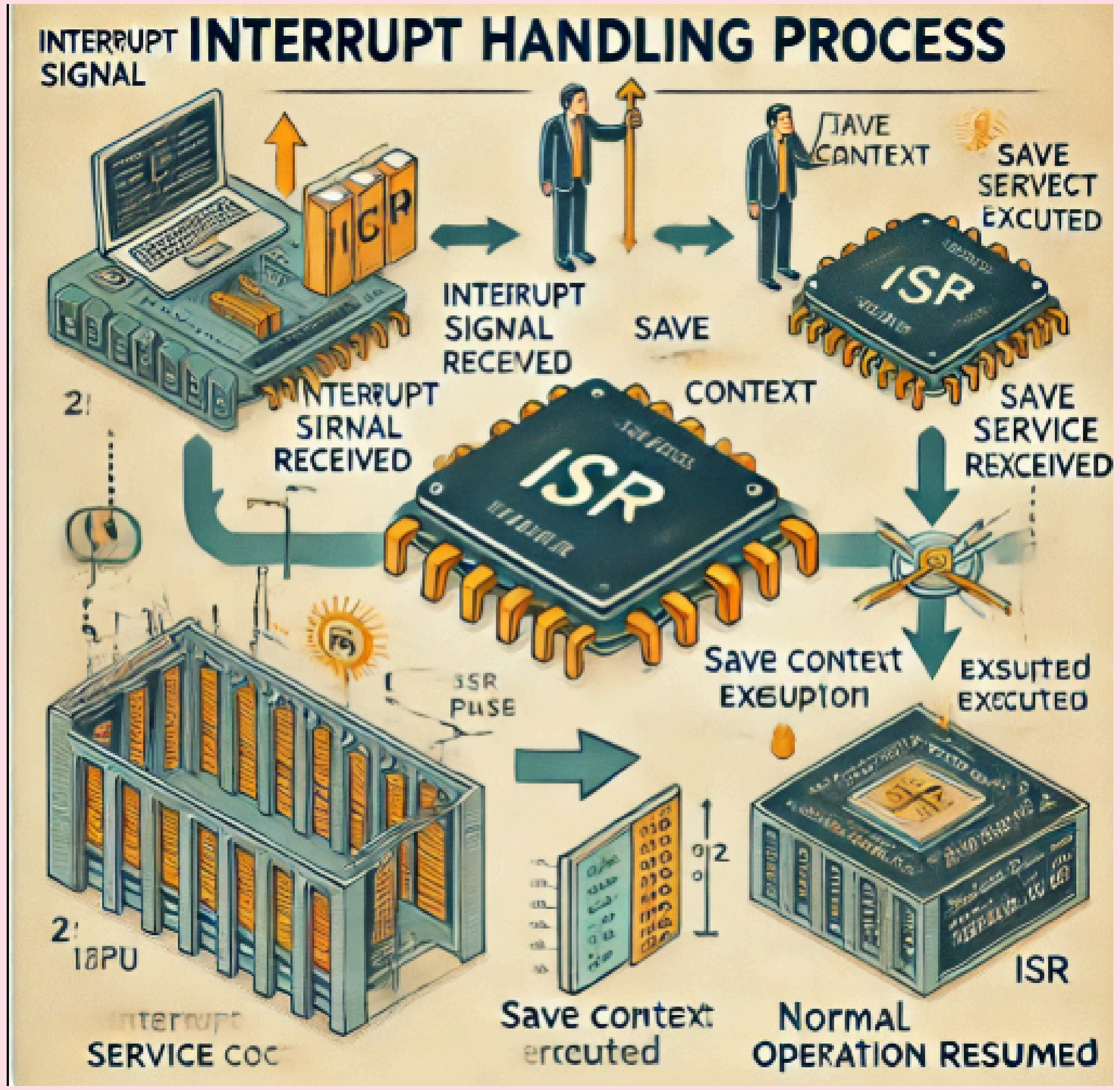
- Step 2:- The current state comprising registers and the program counter is then stored in order to conserve the state of the process.
- Step 3:- The current interrupt and its handler is identified through the interrupt vector table in the processor.
- Step 4:- This control now shifts to the interrupt handler, which is a function located in the kernel space.
- Step 5:- Specific tasks are performed by Interrupt Service Routine (ISR) which are essential to manage interrupt.
- Step 6:- The status from the previous session is retrieved so as to build on the process from that point.
- Step 7:- The control is then shifted back to the other process that was pending and the normal process continues.

EXAMPLE: INTERRUPT HANDLING IN ARDUINO

```
const int buttonPin = 2;     // Interrupt pin (D2 on Arduino UNO)
const int ledPin = 13;       // LED pin (D13 on Arduino UNO)
volatile int buttonPressCount = 0;
unsigned long lastInterruptTime = 0;  // Last time the interrupt was triggered
unsigned long debounceDelay = 200;    // Debounce time in milliseconds
void countButtonPresses() {
  unsigned long interruptTime = millis();  // Get the current time
  if (interruptTime - lastInterruptTime > debounceDelay) {
    buttonPressCount++;  // Increment the count
    lastInterruptTime = interruptTime;  // Update the last interrupt time
  }
}

void setup() {
  Serial.begin(9600);
  pinMode(buttonPin, INPUT_PULLUP);

  // Configure the LED pin as output
  pinMode(ledPin, OUTPUT);
  digitalWrite(ledPin, LOW); // Ensure LED is off initially
  attachInterrupt(digitalPinToInterrupt(buttonPin), countButtonPresses, FALLING);
  Serial.println("Press the button to count...");
}

void loop() {
  Serial.print("Button Press Count: ");
  Serial.println(buttonPressCount);
  if (buttonPressCount % 2 == 0) {
    digitalWrite(ledPin, HIGH); // Turn LED on if count is even
  } else {
    digitalWrite(ledPin, LOW);  // Turn LED off if count is odd
  }
  delay(500);
}
```



Interrupt Handling Process



Step-by-step Flow:

1. CPU receives an interrupt signal.
2. Current execution is paused (context saved).
3. ISR is executed.
4. CPU restores context and resumes normal operation.

Top-half(Hard IRS° and Bottom-half (Soft IRQ, Tasklets)

1. Top-half (Hard IRQ)

The Top-half, also referred to as Hard IRQ, is the first part of the interrupt handling process. It is responsible for the immediate response to the interrupt request (IRQ). The Top-half handles the critical, time-sensitive work that needs to be done as soon as the interrupt occurs.

Characteristics of the Top-half (Hard IRQ):

- Fast Execution: The Top-half is designed to be executed as quickly as possible because it handles the interrupt request immediately.
- Direct Execution: Only the minimum necessary tasks are performed here, such as reading data from hardware or switching the state of the processor.
- Interrupts Disablement: In many systems, while the Top-half is executing, other interrupts (of the same or lower priority) are often disabled to prevent interference.

Goal of the Top-half:

- Quickly respond to the interrupt.
- Ensure that the system can return to normal operation without delays.
- Only critical work is done here to ensure fast processing of the interrupt.

. Example of Top-half (Hard IRQ):

In embedded systems like AVR microcontrollers, the Top-half might look like this:

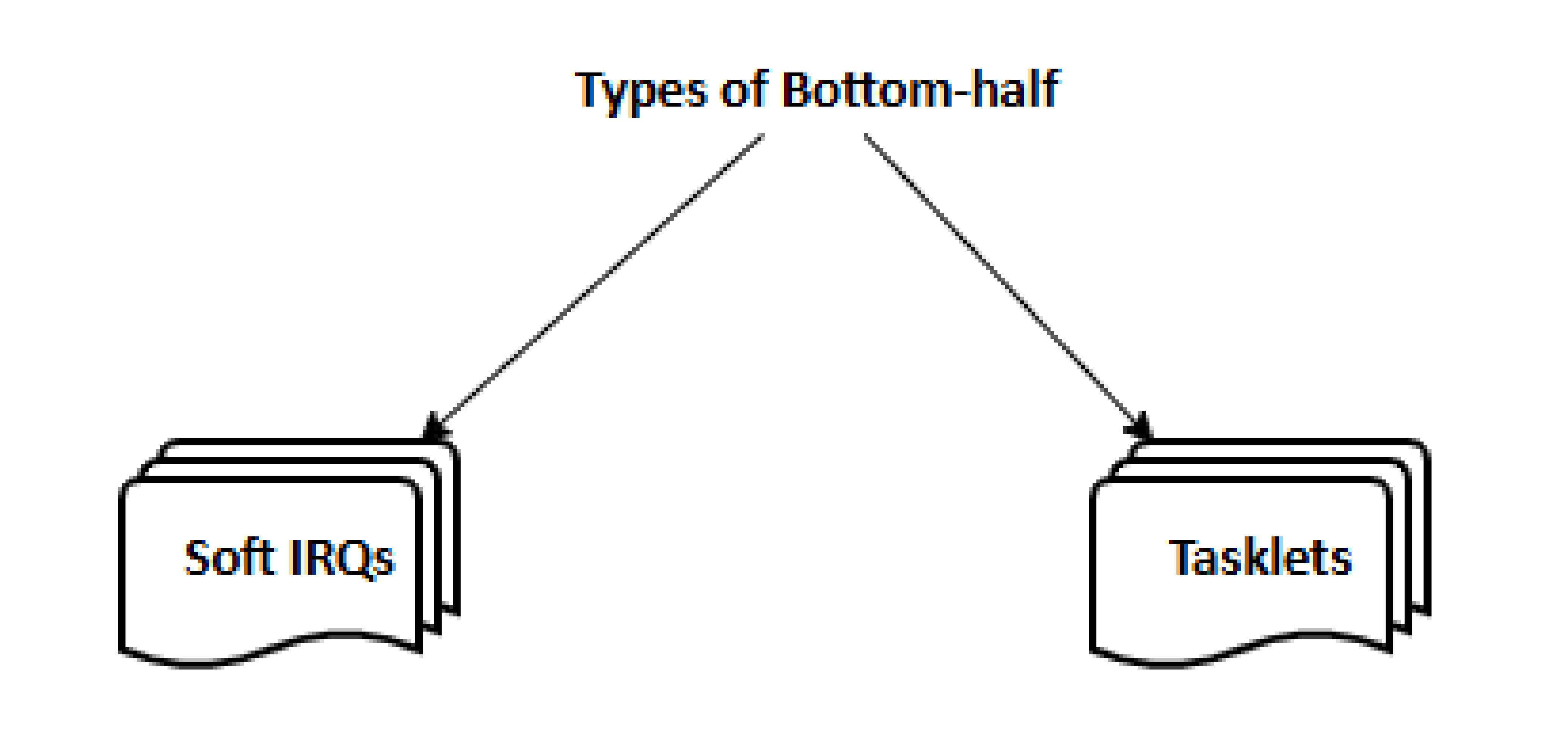
2. Bottom-half (Soft IRQ, Tasklets)

The Bottom-half is the second part of the interrupt handling process, and it is used to deal with non-urgent work that can be deferred. After the Top-half finishes executing, the Bottom-half takes over and processes tasks that were delayed to prevent slowing down the interrupt handling.

Characteristics of the Bottom-half (Soft IRQ, Tasklets):

- Non-urgent: Tasks in the Bottom-half do not need to be executed immediately and can be deferred until the system is not busy handling high-priority tasks.
- Non-critical work: This section deals with tasks like updating data structures or scheduling other tasks, which are not time-sensitive.
- Executed outside interrupt context: The Bottom-half can be executed later, outside the immediate interrupt context, which allows for a smoother execution flow.

Types of Bottom-half:



1.Soft IRQs: These are used to handle tasks that can be delayed a little while ensuring the system remains responsive.

2.Tasklets: These are similar to Soft IRQs but typically involve slightly more overhead. Tasklets are used in kernel systems like Linux to handle deferred work.

Goal of the Bottom-half:

- To minimize the time spent in the interrupt context (Top-half).
- To handle tasks that are less urgent and can be deferred without affecting system responsiveness.

To optimize processor and resource utilization by performing less critical tasks after the interrupt handling.

Example of Bottom-half (Soft IRQ / Tasklets):

```c
#include <linux/interrupt.h>

// Define a Tasklet
static struct tasklet_struct my_tasklet;

void my_tasklet_handler(unsigned long data) {
    // Bottom-half: Processing non-urgent work
    printk(KERN_INFO "Processing deferred task\n");
}

void irq_handler(int irq, void *dev_id, struct pt_regs *regs) {
    // Top-half: Handle the interrupt
    tasklet_schedule(&my_tasklet);  // Schedule Tasklet for Bottom-half
}

int init_module() {
    // Initialize the Tasklet
    tasklet_init(&my_tasklet, my_tasklet_handler, 0);
    return 0;
}

void cleanup_module() {
    // Clean up the Tasklet when the module is unloaded
    tasklet_kill(&my_tasklet);
}
```

- irq_handler: This function handles the interrupt (Top-half). After performing the urgent tasks, it schedules a Tasklet to run later (Bottom-half).
- my_tasklet_handler: This is the non-urgent task (Bottom-half), which is executed after the interrupt has been handled.

Difference Between Top-half and Bottom-half:

Aspect	Top-Half (Hard IRQ)	Bottom-Half (Soft IRQ, Tasklets)
Execution Context	Interrupt context	Kernel thread context
Priority	High	Lower
Task Complexity	Minimal	Complex
Latency Impact	Immediate	Deferred

Benefits of Interrupt

- Real-time Responsiveness: Interrupts permit a system to reply promptly to outside events or signals, permitting real-time processing.
- Efficient Resource usage: Interrupt-driven structures are more efficient than system that depend on busy-waiting or polling strategies. Instead of continuously checking for the incidence of event, interrupts permit the processor to remain idle until an event occurs, conserving processing energy and lowering energy intake.

Multitasking and Concurrency: Interrupts allow multitasking with the aid of allowing a processor to address multiple tasks concurrently.

conclusion

In this lesson, we learned about interrupt management in systems. Interrupts allow systems to quickly respond to important events.
Top-half (Hard IRQ) handles urgent tasks immediately when an interrupt occurs.
Bottom-half (Soft IRQ, Tasklets) processes non-urgent tasks after the Top-half.
By dividing interrupt handling into two parts, the system can quickly address interrupts while ensuring overall efficiency and improved performance.

reference

1-https://www.geeksforgeeks.org/interrupts/
2-https://binaryterms.com/interrupts-in-computer-architecture.html#TypesofInterrupts
3-https://www.researchgate.net/profile/Javad-Ebrahimian-Amiri-2/publication/308843408_A_predictable_interrupt_management_policy_for_real-time_operating_systems/links/5ea2ad16a6fdcc88fc3a28e1/A-predictable-interrupt-management-policy-for-real-time-operating-systems.pdf

Thank you for listening