I/O and Interrupt Handling

I/O (Input / Output)

1. Synchronous I/O

- In synchronous I/O, the **CPU waits** until the I/O operation is completed.
- During this time, the CPU remains blocked and cannot perform other tasks.

Hardware Technique: Polling

- The CPU repeatedly checks the status of the I/O device until it becomes ready.

---

2. Asynchronous I/O

- In asynchronous I/O, the **CPU does not wait** for the I/O operation to complete.
- The CPU continues executing other tasks while the I/O operation is handled in parallel.

Hardware Technique: Interrupts

- The I/O device notifies the CPU when the operation is complete.

Interrupt Processing

- An I/O event is detected by the **I/O device controller**.
- The event is conveyed to the CPU as a special signal called an **Interrupt**.
- The CPU:
    1. Pauses the current execution
    2. Fetches the address of the **Interrupt Service Routine (ISR)** from the **Interrupt Vector Table (IVT)**
    3. Executes the ISR
    4. Resumes execution from the point where it was paused

Hardware Interrupt vs Software Interrupt

Hardware Interrupt

- Generated by **hardware peripherals**
- Examples: Timer overflow, GPIO event, UART receive

Software Interrupt

- Generated by executing a **special instruction** at assembly or machine level

- When executed:
  - Current execution is paused
  - Interrupt handler is executed
  - Execution resumes after handler completion

Architecture-Specific Names:

- **8085 / 8086:** `INT`
- **ARM7:** `SWI` (Software Interrupt)
- **ARM Cortex-M:** `SVC` (Supervisor Call)

> Software interrupts are also called **Traps** in some architectures.

Interrupt Controller

- Responsible for conveying interrupt requests from multiple peripherals to the CPU.
- Manages **interrupt priority** when multiple interrupts occur simultaneously.

Examples:

- **8085 / 8086:** 8259 PIC
- **Modern x86:** APIC
- **ARM7:** VIC (Vectored Interrupt Controller)
- **ARM Cortex-M3/M4:** NVIC (Nested Vectored Interrupt Controller)

Interrupt Delivery in ARM Processors

- ARM processors use an **Interrupt Controller** to:
  - Receive interrupt requests from peripherals
  - Resolve priorities
  - Signal the CPU core

General Flow:

1. The interrupt controller receives interrupt requests.
2. It resolves priority among pending interrupts.
3. The highest-priority interrupt is signaled to the CPU.
4. The CPU:
   - Saves its current state
   - Identifies the interrupt source
   - Executes the corresponding ISR from the vector table
5. After servicing:
   - The ISR signals completion
   - The CPU resumes normal execution or services the next interrupt

---

The Interrupt Delivery Process

1. Interrupt Assertion

- A peripheral device generates an interrupt request.
- The request is sent to the interrupt controller.

---

2. Interrupt Controller Actions

- **Configuration:** The controller is configured to manage interrupts.
- **Priority Resolution:** Determines the highest-priority pending interrupt.
- **Signaling the CPU:** Forwards the selected interrupt to the CPU interface.

---

3. Core Interaction

- **Interrupt Signal:** CPU interface notifies the core.
- **Entering Exception Mode:**
  - Current task is suspended
  - CPU context (registers) is saved
- **Vector Table Lookup:**
  - CPU jumps to the corresponding entry in the interrupt vector table

---

4. Interrupt Handler Execution

- **Interrupt Identification:**
  - Top-level handler reads the interrupt ID from the CPU interface register
- **Dispatching the Handler:**
  - Calls the device-specific ISR
- **Servicing the Interrupt:**
  - ISR performs required operations to handle the interrupt

---

5. Interrupt Completion

- **Signaling Completion:**
  - ISR writes the interrupt ID to the **End Of Interrupt (EOI)** register
- **Status Update:**
  - Interrupt controller marks the interrupt as inactive
- **Resuming Operation:**
  - CPU restores previous context
  - Execution resumes from the interrupted task or processes the next interrupt

---

`

`

`

`

STM32 / NVIC-specific mapping

STM32 / ARM Cortex-M NVIC Specific Mapping

STM32 microcontrollers are based on the **ARM Cortex-M** architecture, which uses the **NVIC (Nested Vectored Interrupt Controller)** for interrupt management.

NVIC Overview

- NVIC is **integrated inside the Cortex-M core** (not an external chip).
- It handles:
  - Interrupt enabling / disabling
  - Priority assignment
  - Nested interrupts
  - Vectoring to the correct ISR

STM32 Interrupt Sources

Interrupts in STM32 can originate from:

- GPIO (EXTI lines)
- Timers (TIMx)
- UART / USART
- SPI / I2C
- ADC
- DMA
- System exceptions (SysTick, HardFault, etc.)

Each interrupt source has a **fixed interrupt number (IRQn)**.

Vector Table in STM32

- The **vector table** is located at the start of Flash memory by default:

0x0800 0000

- It contains:

1. Initial Stack Pointer value
2. Reset Handler address
3. Exception handlers
4. Peripheral interrupt handlers

Example:

- EXTI0 → `EXTI0_IRQHandler`
- TIM2 → `TIM2_IRQHandler`

- USART2 → `USART2_IRQHandler`

NVIC Registers (Conceptual)

The NVIC provides memory-mapped registers to control interrupts:

- **ISER** – Interrupt Set Enable Register
- **ICER** – Interrupt Clear Enable Register
- **ISPR** – Interrupt Set Pending Register
- **ICPR** – Interrupt Clear Pending Register
- **IPR** – Interrupt Priority Register

These registers are accessed using **memory-mapped I/O**.

Interrupt Priority in STM32

- STM32 supports **priority levels** (lower number = higher priority).
- NVIC allows:
- Preemption priority
- Subpriority (device-dependent)

Example:

- Timer interrupt can preempt UART interrupt
- GPIO interrupt can be lowest priority

Interrupt Entry Sequence (STM32)

When an interrupt occurs:

1. NVIC receives the interrupt request.
2. NVIC checks:
    - Interrupt enable status
    - Priority level
3. NVIC signals the Cortex-M core.
4. Cortex-M automatically:
    - Pushes registers (R0–R3, R12, LR, PC, xPSR) onto stack
    - Switches to Handler mode
5. CPU jumps to ISR address from vector table.

Interrupt Exit Sequence (STM32)

1. ISR completes execution.
2. `BX LR` instruction triggers exception return.
3. Cortex-M:
    - Pops saved registers from stack

    - Restores previous execution state
4. Normal program execution resumes.

EXTI + NVIC Example Mapping (Conceptual)

- GPIO Pin → EXTI Line
- EXTI Line → NVIC IRQ Number
- NVIC IRQ → ISR Function

Example:

- PA0 → EXTI0 → `EXTI0_IRQn` → `EXTI0_IRQHandler()`

Key Points

- NVIC is **core-integrated**, not peripheral-based.
- Interrupt latency is very low due to **hardware stacking**.
- Cortex-M supports **nested interrupts**.
- ISR address is fetched directly from the vector table.
- STM32 uses **memory-mapped NVIC registers**.

Weak Functions, External Interrupts & Volatile Concept (STM32 / ARM Cortex-M)

Weak Functions (GCC)

- Weak functions are supported in **both Assembly and C** using the **GCC toolchain**.
- A weak function acts as a **temporary (default) implementation**.
- It is typically used to provide a default handler that can be overridden by the user.

Behavior of Weak Functions

- If **no function** with the same name is implemented elsewhere, the **weak function** is invoked.
- If a function is implemented with the **exact same name**, the strong (user-defined) function overrides the weak one.

Typical Usage in Embedded Systems

- Default interrupt handlers
- Library-provided hooks
- Startup code (e.g., `Default_Handler`)

`

External Interrupt (EXTI) – STM32

External Interrupt Features

- All GPIO pins can be configured to generate interrupts on:
  - Rising edge
  - Falling edge
  - Rising & Falling edge
- STM32 External Interrupt Controller supports **up to 23 edge detectors**.
- GPIO pin mapping:
  PORTx.n → EXTIn (0 ≤ n ≤ 15)
- EXTI lines:
- **EXTI0–EXTI15** → GPIO pins
- **EXTI16–EXTI22** → RTC, USB, Ethernet, etc.

EXTI Registers

EXTI Peripheral Registers

- **IMR** – Interrupt Mask Register
- `0` → Masked
- `1` → Not masked
- **EMR** – Event Mask Register
- `0` → Event masked
- `1` → Event enabled
- **RTSR** – Rising Trigger Selection Register
- `0` → Rising edge disabled
- `1` → Rising edge enabled
- **FTSR** – Falling Trigger Selection Register
- `0` → Falling edge disabled
- `1` → Falling edge enabled
- **PR** – Pending Register
- `0` → Not pending
- `1` → Pending

- Writing `1` clears the pending interrupt

SYSCFG Registers (EXTI Mapping)

Used to connect GPIO ports to EXTI lines.

- **EXTICR1** → EXTI0–3
- **EXTICR2** → EXTI4–7
- **EXTICR3** → EXTI8–11
- **EXTICR4** → EXTI12–15

Each EXTI line uses **4 bits** to select the GPIO port:
PA = 0, PB = 1, PC = 2, ... PI = 8

NVIC Registers (Conceptual)

- **ISERx** – Interrupt Set Enable Register
  - HAL equivalent: `NVIC_EnableIRQ(irq);`

NVIC handles:

- Interrupt enabling/disabling
- Priority management
- Nested interrupts

External Interrupt Configuration Steps

1. Configure GPIO pin as **input** (pull-up / pull-down as required).
2. Map GPIO pin to EXTI line using **SYSCFG EXTICR**.
3. Configure trigger type:
   - Rising edge → RTSR
   - Falling edge → FTSR
4. Unmask interrupt in **IMR**.
5. Enable interrupt in **NVIC**.
6. Implement interrupt handler.
7. Acknowledge (clear) interrupt pending flag.
8. Keep ISR **short and non-blocking**.

> ISR function name **must match** the name in the interrupt vector table.

Interrupt Handling Guidelines

- While an ISR is executing:
  - All **lower-priority interrupts are masked**
  - This increases interrupt latency
- ISR should be:

    - Short
    - Non-blocking
- Any blocking or heavy processing must be deferred to:
    - Main loop
    - Worker function
    - RTOS task (if used)

Optimization Effect on Interrupt Code

Optimization Level: `-O 0`

```
uint32_t ext_flag;

void EXTI0_IRQHandler(void)
{
    // Acknowledge interrupt
    EXTI->PR |= BV(0);

    // Set flag
    ext_flag = 1;
}

while (ext_flag == 0)
    ;
ext_flag = 0;
```

- Assembly behavior:
    - Variable is repeatedly read from memory
- Works as expected

Optimization Level: -O3

```
uint32_t ext_flag;

void EXTI0_IRQHandler(void)
{
    EXTI->PR |= BV(0);
    ext_flag = 1;
}

while (ext_flag == 0)
    ;

ext_flag = 0;
```

- Assembly behavior:
    - Compiler may cache ext_flag in a CPU register
    - ISR update may be ignored
    - Results in infinite loop

Register Caching and volatile Keyword

Compiler may cache frequently used variables in CPU registers for optimization.

- If a variable is modified:
    - Inside an ISR
    - By hardware (MMIO)
    - By another execution context the cached value becomes invalid.

Solution: volatile

- volatile forces the compiler to:
    - Always read the variable from memory
    - Disable register caching
- Rule
    - Variables modified by:
    - ISR
    - Hardware registers
    - DMA
- must be declared volatile

Interrupt Handler as a Separate Execution Context

- ISR is a separate thread of execution
- It can modify shared variables asynchronously
- Such shared variables must be declared volatile

Memory-Mapped I/O

- Peripheral registers are mapped to fixed memory addresses.
- Accessed like normal variables using pointers.
- Can change outside program control.

static, const, and volatile Keywords

- static
    - Limits scope to file or function
    - Stored in data section
    - Value can be modified
- const
    - Value cannot be modified in source code
    - Compiler enforces read-only behavior

- volatile
    - Disables optimization
    - Always reads value from memory
    - Required for MMIO and ISR-shared variables
- Example :

```
static const volatile int i = 1;
```

- Meaning:
    - static → scoped to file/function
    - const → cannot be modified in source code
    - volatile → value may change externally (MMIO / ISR)
- Typical use case:
    - Memory-mapped hardware registers

Key Takeaways

- Weak functions allow safe default implementations.
- EXTI + NVIC manage GPIO-based interrupts in STM32.
- ISR must be short and non-blocking.
- Compiler optimizations can break interrupt-based logic.
- volatile is mandatory for ISR-shared and MMIO variables.