Department of Electrical & Computer Engineering
ENCS4110 – Computer Design Laboratory

Experiment 05

GPIO Inputs and Interrupt Handling

Date: May 2026

5 GPIO Inputs and Interrupt Handling

Learning Objectives

After completing this experiment, you will be able to:

- Configure GPIO inputs with digital enable and internal pull resistors.
- Read switches via polling with software debouncing.
- Configure edge-triggered GPIO interrupts and write minimal ISRs.
- Enable and manage interrupts in the NVIC (IRQ mapping, ISER).
- Unlock and configure protected pins (PF0/NMI) for general-purpose use.

Experiment Overview

This experiment extends GPIO functionality to inputs and introduces interrupt-driven programming. You will configure GPIO pins to read mechanical switches using both polling and interrupt approaches, implement software debouncing techniques, write interrupt service routines, and enable GPIO interrupts in the NVIC. By the end of this lab, you will understand how to implement both polling and interrupt-driven input handling, and write responsive embedded applications that react to external events in real time.

5.1 Theoretical Background

5.1.1 GPIO Input Configuration

In the previous experiment, GPIO pins were configured as outputs to control LEDs. In this experiment, we configure GPIO pins as **inputs** to read the state of external devices such as switches, buttons, and sensors.

When a GPIO pin is configured as an input, the microcontroller reads the voltage level on the pin and interprets it as a logic '0' (low, typically 0V) or logic '1' (high, typically 3.3V). The pin's digital input buffer must be enabled, and the pin must be connected to a defined voltage level to avoid floating states.

5.1.1.1 Input Pin Requirements

For reliable digital input operation, three conditions must be met:

- **Direction**: The pin must be configured as an input (bit cleared in `GPIODIR`).
- **Digital Enable**: The digital input buffer must be enabled (bit set in `GPIODEN`).
- **Defined Logic Level**: The pin must be connected to a valid logic level (not floating).

If a pin is left floating (not connected to a defined voltage), it can pick up electrical noise and produce random or unstable readings. To prevent this, pins are typically connected to either power (`VCC`) or ground (`GND`) through a resistor, or the microcontroller's internal pull-up or pull-down resistors can be enabled.

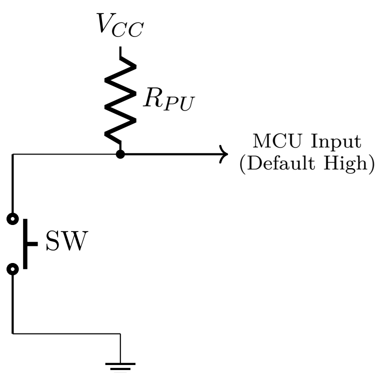
5.1.1.2 Pull-Up and Pull-Down Resistors

Digital inputs must not be left floating: an undefined voltage can produce random logic reads. A pull resistor defines the idle (no-press) level of a switch input.

Pull-Up (active-low) A **pull-up** connects the pin weakly to V_{CC}, so the idle level is logic 1. Pressing the switch drives the pin to GND → logic 0.

Pull-Down (active-high) A **pull-down** connects the pin weakly to GND, so the idle level is logic 0. Pressing the switch drives the pin to V_{CC} → logic 1.

A) Pull-Up Configuration



B) Pull-Down Configuration

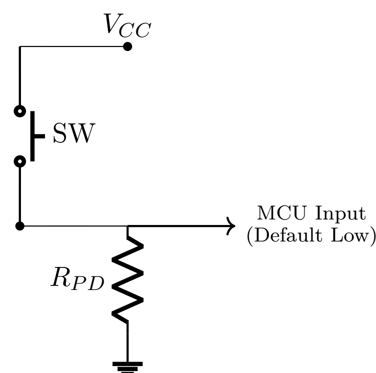


Figure 5.1: Pull-Up and Pull-Down Resistor Configurations for Switch Inputs

Internal Resistors (TM4C123) TM4C123 GPIO pins include configurable **internal** pull resistors:

- `GPIOPUR` — Pull-Up Select (enable bit = pull-up on that pin)
- `GPIOPDR` — Pull-Down Select (enable bit = pull-down on that pin)

Do not enable both on the same pin. On the LaunchPad, SW1 (PF4) and SW2 (PF0) short to GND when pressed, so enable **pull-ups** (`GPIOPUR`) to make them active-low (read 0 when pressed, 1 when released).

5.1.2 Switch Bouncing and Debouncing

When a mechanical switch or button is pressed or released, the metal contacts inside do not make or break contact cleanly. Instead, they **bounce** — rapidly making and breaking contact multiple times before settling into a stable state. This bouncing typically lasts 5-30 milliseconds.

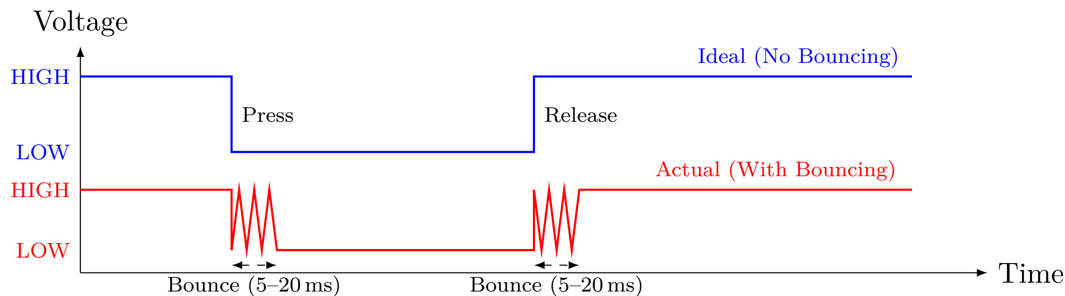


Figure 5.2: Mechanical Switch Bounce — Press and Release Both Exhibit Bouncing

For applications that poll the input at a slow rate or only care about the final state, bouncing may not be a problem. However, for interrupt-driven systems or applications that count button presses, bouncing can cause multiple false triggers.

5.1.2.1 Debouncing Techniques

Two approaches can eliminate or mitigate switch bouncing:

Hardware Debouncing Add an RC (resistor-capacitor) filter circuit or a dedicated debouncing IC to the switch. The capacitor smooths out the voltage transitions, preventing bounces from reaching the microcontroller input.

Software Debouncing After detecting a button press (or release), wait for a short period (typically 10-50 ms) to allow the bouncing to settle, then read the input again to confirm the state. This can be implemented with:

- **Blocking delay debouncing**: Insert a fixed delay (busy-wait loop) after detecting an edge, then verify the input state. This is simple but prevents the CPU from doing other work during the delay period.
- **Timer-based debouncing (non-blocking)**: Use a hardware timer to sample the input periodically without blocking the main program. Only register a press after multiple consistent readings across timer intervals.

For this experiment, we will implement simple delay-based debouncing in our polling examples. More sophisticated timer-based debouncing techniques will be covered in the following experiment.

5.1.3 Reading GPIO Inputs: Polling vs. Interrupts

There are two fundamental approaches to reading digital inputs:

5.1.3.1 Polling (Continuous Checking)

In polling, the CPU continuously reads the input pin in a loop and checks whether the state has changed. This is simple to implement but has drawbacks:

- **CPU Usage**: The CPU is busy checking the input even when nothing is happening.
- **Latency**: The response time depends on how frequently the loop runs.
- **Power Consumption**: The CPU cannot enter low-power sleep modes while polling.

Polling is suitable for simple applications where the CPU has nothing else to do or when inputs change slowly.

```
while (1) {
    if (button is pressed) {
        // Respond to button press
    }
}
```

Listing 5.1: Polling Example (Conceptual)

5.1.3.2 Interrupts (Event-Driven Response)

In interrupt-driven input, the GPIO peripheral notifies the CPU when an input change occurs by generating an **interrupt request** (IRQ). The CPU immediately stops its current task, executes an **Interrupt Service Routine** (ISR), and then resumes normal operation.

- **Efficiency**: The CPU can do other work or sleep until an input event occurs.
- **Responsiveness**: The CPU reacts immediately to input changes.
- **Power Efficiency**: The CPU can remain in low-power modes and wake only when needed.

Interrupts are essential for responsive embedded systems and are the preferred method for handling asynchronous events.

```
// Main loop can perform other tasks
while (1) {
    // Do other work or sleep
}

// ISR is called automatically when button is pressed
void ButtonISR(void) {
    // Respond to button press
    // Clear interrupt flag
}
```

Listing 5.2: Interrupt Example (Conceptual)

5.1.4 How Interrupts Work

Understanding the interrupt mechanism is essential for writing interrupt-driven programs. This section explains the complete lifecycle of an interrupt from trigger to completion.

5.1.4.1 Interrupt Lifecycle

When a GPIO pin configured for interrupts detects the specified event (e.g., a button press), the following sequence occurs:

1. Event Detection The GPIO peripheral continuously monitors the pin according to its configuration (edge/level detection). When the configured condition is met (e.g., falling edge), the peripheral sets an internal flag.

2. Interrupt Request (IRQ) If the interrupt is unmasked (enabled in `GPIOIM`), the GPIO module sends an interrupt request to the NVIC. The NVIC receives interrupt requests from all peripherals and manages their execution.

3. CPU Response The NVIC checks if the interrupt is enabled (`NVIC_ISER`) and compares its priority with currently executing code. If the interrupt should be serviced:

- The CPU finishes executing the current instruction.
- The CPU automatically saves the current execution context (program counter, registers, status flags) onto the stack.
- The CPU loads the address of the Interrupt Service Routine (ISR) from the vector table.
- Execution jumps to the ISR.

4. ISR Execution The ISR (e.g., `GPIOF_Handler()`) executes and must:

- Identify which pin(s) caused the interrupt (read `GPIOMIS`).
- Perform the required response (toggle LED, set flag, etc.).
- Clear the interrupt flag (write to `GPIOICR`) — **critical step**.

5. Return from Interrupt When the ISR completes (returns), the CPU:

- Automatically restores the saved context from the stack.
- Resumes execution from where it was interrupted.

This entire process happens in microseconds, making interrupts extremely responsive.

5.1.4.2 Why Clear the Interrupt Flag?

The interrupt flag remains set until explicitly cleared by software. If the ISR does not clear the flag by writing to `GPIOICR`, the NVIC will immediately re-trigger the interrupt as soon as the ISR returns, creating an infinite loop of interrupts that hangs the system.

```c
void GPIOF_Handler(void) {
    // Read which pin caused interrupt
    if (GPIOF->MIS & (1 << 4)) {
        // Handle PF4 interrupt
        // ...

        GPIOF->ICR |= (1 << 4);  // MUST clear flag!
    }
}
```

Listing 5.3: Critical: Always clear the interrupt flag

5.1.4.3 Interrupt Advantages and Considerations

Advantages:

- **Efficient**: CPU can perform other tasks or sleep while waiting for events.
- **Responsive**: Immediate response to external events (microsecond latency).
- **Power-efficient**: CPU can remain in low-power modes between events.
- **Real-time**: Predictable timing for critical events.

Considerations:

- **Complexity**: More complex to implement and debug than polling.
- **Shared data**: ISRs and main code must carefully manage shared variables (use `volatile`).
- **ISR constraints**: ISRs should be short and fast — avoid delays and lengthy operations.
- **Debouncing**: Mechanical switch bouncing can trigger multiple interrupts; software or hardware debouncing is often necessary.

5.1.5 GPIO Input Configuration Registers

Before a GPIO pin can be used as an input, it must first be configured for digital functionality. As discussed in the previous experiment, the following three steps are required for **all GPIO pins**:

1. **Enable the GPIO clock** via `SYSCTL_RCGCGPIO`.
2. **Select the direction** of the pin using `GPIO_DIR` (`0` for input, `1` for output).
3. **Enable digital functionality** using `GPIO_DEN`.

Once a pin is set as an input, the logic level on the pin may be undefined if it is left floating (not driven by an external source). To ensure a stable logic level, internal resistors can be activated: These resistors are controlled by two dedicated registers:

- `GPIO_PUR` — Enable internal pull-up resistor on the selected pin(s).
- `GPIO_PDR` — Enable internal pull-down resistor on the selected pin(s).

GPIOPUR — Pull-Up Resistor Enable

Register: `GPIOPUR` — GPIO Pull-Up Select (Port F: `0x40025510`)
The `GPIOPUR` register enables the internal pull-up resistor for each GPIO pin. When enabled, the pin is weakly pulled to logic '1' (3.3V) through a resistor (typically 13-40 kω). This prevents floating inputs and is essential for switches connected to ground.

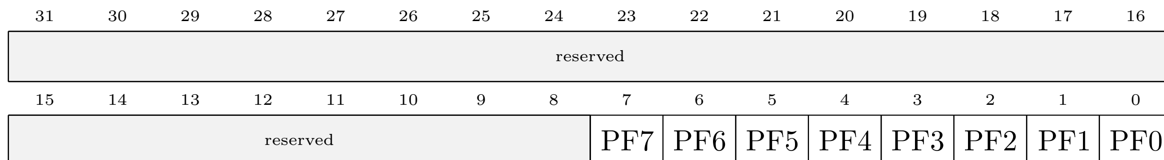


Figure 5.3: GPIOPUR Register — Pull-Up Enable

Bit Values:

- **0**: Pull-up disabled (default)
- **1**: Pull-up enabled

For the LaunchPad switches (SW1 on PF4, SW2 on PF0), pull-ups must be enabled.

GPIOPDR — Pull-Down Resistor Enable

Register: `GPIOPDR` — GPIO Pull-Down Select (Port F: `0x40025514`)
The `GPIOPDR` register enables the internal pull-down resistor for each GPIO pin. When enabled, the pin is weakly pulled to logic '0' (0V) through a resistor. This is used for switches connected to power (`VCC`).

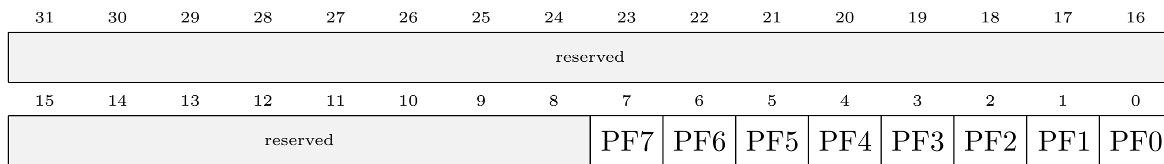


Figure 5.4: GPIOPDR Register — Pull-Down Enable

Bit Values:

- **0**: Pull-down disabled (default)
- **1**: Pull-down enabled

Note: Pull-up and pull-down resistors should not be enabled simultaneously on the same pin.

5.1.6 Unlocking Protected GPIO Pins

Some GPIO pins on the TM4C123 are **locked** by default to prevent accidental reconfiguration of critical functions such as JTAG/SWD (debug interface) and NMI (Non-Maskable Interrupt).

On Port F, pin **PF0** is locked because it is the NMI input by default. To use PF0 as a general-purpose GPIO input (for SW2 on the LaunchPad), it must be unlocked.

GPIOLOCK — Lock Register

Register: `GPIOLOCK` — GPIO Lock (Port F: `0x40025520`)
The `GPIOLOCK` register controls write access to the `GPIOCR` register. Writing the magic value `0x4C4F434B` ("LOCK" in ASCII) unlocks the port and allows modifications to the commit register.
Values:

- Write `0x4C4F434B`: Unlock the port (allow changes to `GPIOCR`)
- Write any other value: Lock the port (prevent changes to `GPIOCR`)
- Read: Returns `0x00000001` if locked, `0x00000000` if unlocked

GPIOCR — Commit Register

Register: `GPIOCR` — GPIO Commit (Port F: `0x40025524`)
The `GPIOCR` register controls which pins can be reconfigured. After unlocking with `GPIOLOCK`, set the corresponding bit in `GPIOCR` to allow changes to that pin's configuration registers.

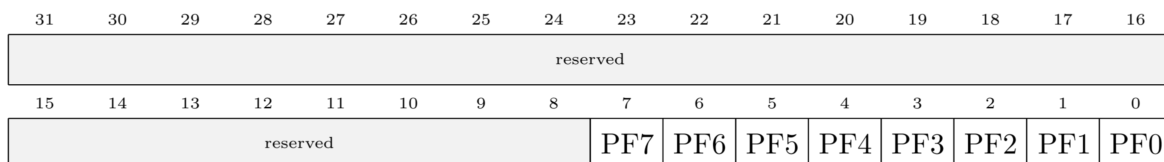


Figure 5.5: GPIOCR Register — Commit Control

Bit Values:

- **0**: Pin configuration is locked (cannot be modified)
- **1**: Pin configuration is unlocked (can be modified)

Unlocking Procedure

To unlock a protected pin (such as PF0):

1. Write the unlock key `0x4C4F434B` to the `GPIOLOCK` register.
2. Set the corresponding bit in the `GPIOCR` (Commit Register) to allow changes.
3. Configure the pin normally (`GPIODIR`, `GPIODEN`, etc.).

```
GPIOF->LOCK = 0x4C4F434B;    // Unlock Port F
GPIOF->CR   |= (1 << 0);     // Allow changes to PF0
```

Listing 5.4: Unlocking PF0 for GPIO use

After unlocking, PF0 can be configured like any other GPIO pin.

5.1.7 GPIO Interrupt Configuration Registers

To configure a GPIO pin to generate interrupts, several registers must be configured. The process involves selecting the trigger condition (edge or level, rising or falling) and enabling the interrupt at both the GPIO module and the NVIC.

GPIOIS — Interrupt Sense Register

Register: `GPIOIS` — GPIO Interrupt Sense (Port F: `0x40025404`)
The `GPIOIS` register determines whether interrupts are triggered by signal **edges** (transitions) or **levels** (steady states).

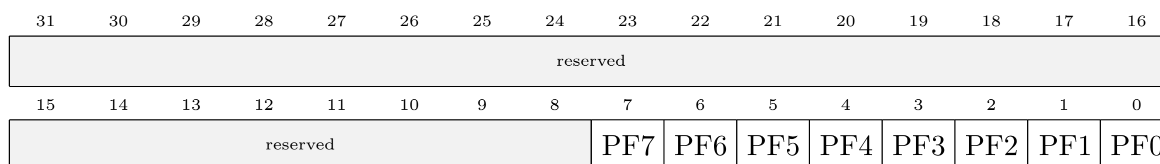


Figure 5.6: GPIOIS Register — Interrupt Sense

Bit Values:

- **0**: Edge-sensitive (detects transitions) — typical for buttons
- **1**: Level-sensitive (detects steady state)

GPIOIBE — Interrupt Both Edges Register

Register: `GPIOIBE` — GPIO Interrupt Both Edges (Port F: `0x40025408`)
When edge-sensitive mode is selected (bit cleared in `GPIOIS`), this register determines whether interrupts occur on a single edge or both edges.

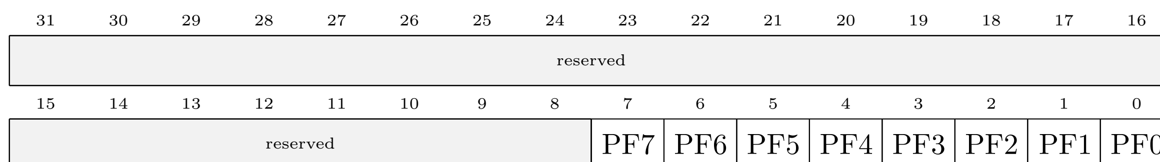


Figure 5.7: GPIOIBE Register — Interrupt Both Edges

Bit Values:

- **0**: Single edge (rising or falling, defined by `GPIOIEV`)

- **1**: Both rising and falling edges trigger interrupts

GPIOIEV — Interrupt Event Register

Register: `GPIOIEV` — GPIO Interrupt Event (Port F: `0x4002540C`)
For single-edge interrupts (when the corresponding bit in `GPIOIBE` is 0), this register selects which edge triggers the interrupt.

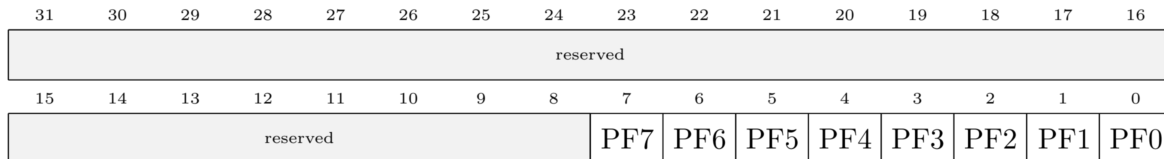


Figure 5.8: GPIOIEV Register — Interrupt Event

Bit Values:
- **0**: Falling edge (high-to-low transition)
- **1**: Rising edge (low-to-high transition)

For switches with pull-ups (LaunchPad SW1/SW2), falling edge interrupts detect button presses.

GPIOIM — Interrupt Mask Register

Register: `GPIOIM` — GPIO Interrupt Mask (Port F: `0x40025410`)
The `GPIOIM` register enables or disables (masks/unmasks) interrupt generation for each pin. Setting a bit to '1' allows that pin to generate interrupts.

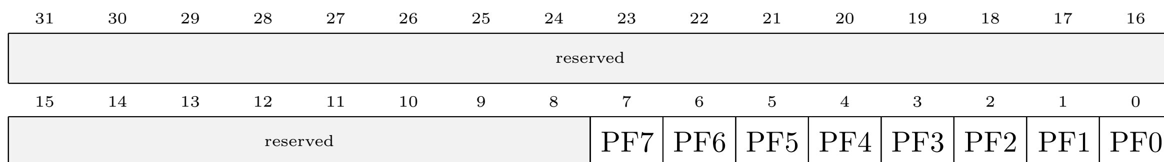


Figure 5.9: GPIOIM Register — Interrupt Mask

Bit Values:
- **0**: Interrupt disabled (masked) — default
- **1**: Interrupt enabled (unmasked)

GPIOMIS — Masked Interrupt Status Register

Register: `GPIOMIS` — GPIO Masked Interrupt Status (Port F: `0x40025418`)
This read-only register shows which pins have pending interrupts after masking. The ISR reads this register to determine which pin(s) caused the interrupt.

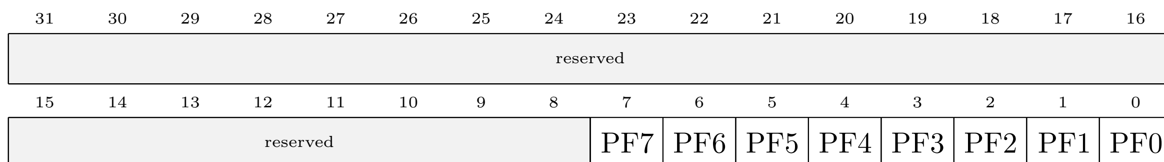


Figure 5.10: GPIOMIS Register — Masked Interrupt Status (Read-Only)

Bit Values:

- **0**: No interrupt pending for this pin
- **1**: Interrupt pending for this pin

GPIOICR — Interrupt Clear Register

Register: `GPIOICR` — GPIO Interrupt Clear (Port F: `0x4002541C`)
Writing '1' to a bit in this register clears the corresponding interrupt flag. This is **critical**: the ISR must clear the flag, or the interrupt will continuously re-trigger.

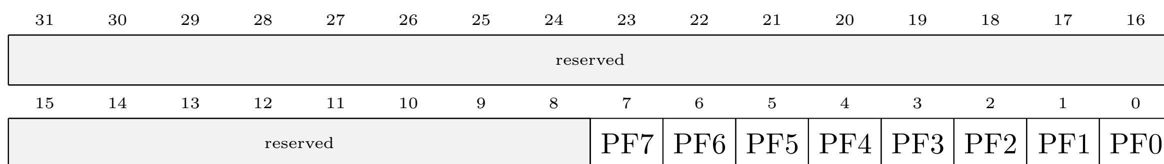


Figure 5.11: GPIOICR Register — Interrupt Clear

Bit Values:

- Write **1** to clear the interrupt flag for that pin
- Write **0** has no effect

5.1.8 Configuration Workflow

GPIO interrupt configuration is divided into two parts, configuring the pin as an input and configuring the interrupt settings. The following steps summarize the process:

Step 1: Pin Configuration Configure the GPIO pin as an input with proper electrical characteristics:

1. Enable the GPIO port clock (`RCGCGPIO`).
2. Unlock protected pins if needed (`GPIOLOCK`, `GPIOCR`).
3. Configure the pin as input (`GPIODIR` = 0).
4. Enable the digital function (`GPIODEN` = 1).
5. Enable pull-up or pull-down resistor (`GPIOPUR` or `GPIOPDR`).

Step 2: Interrupt Configuration

1. Configure interrupt sense (`GPIOIS` = 0 for edge-sensitive).
2. Select edge(s) (`GPIOIBE`/`GPIOIEV`).
3. **Clear any prior flags** (`GPIOICR` = 1 for the pin).
4. Unmask the pin in the GPIO (`GPIOIM` = 1).
5. Enable the IRQ in the NVIC.

5.1.9 NVIC — Nested Vectored Interrupt Controller

The NVIC manages all peripheral interrupts on the Cortex-M4, deciding which ISR to execute based on enable bits, masking, and priority.

NVIC_ISER — Interrupt Set-Enable Registers

Registers: `NVIC_ISER[0..3]` (`0xE000E100-0xE000E10C`)
Each 32-bit `ISER` register enables up to 32 interrupt sources. Writing a '1' to a bit enables that IRQ; writing '0' has no effect.

Register	IRQ Range	Address
`ISER[0]`	0-31	0xE000E100
`ISER[1]`	32-63	0xE000E104
`ISER[2]`	64-95	0xE000E108
`ISER[3]`	96-127	0xE000E10C

Table 5.1: Interrupt ranges for NVIC_ISER registers

For the TM4C123, all GPIO interrupts (IRQ0-IRQ4) for Ports A-E, and Port F (IRQ30) are in `ISER[0]`.

Example — Enabling Port F Manually

```
NVIC->ISER[0] |= (1 << 30);   // IRQ30 - NVIC_EN0 bit 30
```

Listing 5.5: Enable GPIO Port F interrupt manually

CMSIS Alternative (Recommended):

```
NVIC_EnableIRQ(GPIOF_IRQn);
```

This function automatically computes the correct register and bit:

```
void NVIC_EnableIRQ(IRQn_Type IRQn) {
    if (IRQn >= 0)
        NVIC->ISER[IRQn >> 5] |= (1u << (IRQn & 0x1F)); // /32 and %32
}
```

Explanation:

$$\text{Register index} = \left\lfloor \frac{\text{IRQn}}{32} \right\rfloor, \qquad \text{Bit position} = \text{IRQn} \bmod 32.$$

Each `ISER` register enables 32 interrupts. Dividing the IRQ number by 32 selects the correct register index, and the remainder (`IRQn % 32`) identifies the bit position within that 32-bit register.

5.1.10 Interrupt Service Routine (ISR)

When a GPIO interrupt occurs, the NVIC calls the corresponding ISR. For GPIO Port F, the ISR is named `GPIOF_Handler()`. The ISR must:

1. Determine which pin caused the interrupt (read `GPIOMIS`).
2. Perform the required action (e.g., toggle an LED).
3. Clear the interrupt flag (write to `GPIOICR`).

```
void GPIOF_Handler(void) {
    if (GPIOF->MIS & (1 << 4)) {      // Check if PF4 caused interrupt
        // Respond to SW1 press
        GPIOF->ICR |= (1 << 4);       // Clear PF4 interrupt flag
    }
    if (GPIOF->MIS & (1 << 0)) {      // Check if PF0 caused interrupt
```

```
        // Respond to SW2 press
        GPIOF->ICR |= (1 << 0);      // Clear PF0 interrupt flag
    }
}
```

Listing 5.6: GPIO Port F ISR Template

5.2 Procedure

5.2.1 Examples

The following examples demonstrate GPIO input configuration using polling and interrupt-based approaches.

5.2.1.1 Example 1 — Reading Switch Input Using Polling

This example continuously polls the state of SW1 (PF4) and turns on the red LED (PF1) when the button is pressed.

```
#include "TM4C123.h"

#define RED_LED 0x02
#define SWITCH 0x10

int main(void) {
    unsigned int state;
    SYSCTL->RCGCGPIO |= (1<<5);        // Enable Port F
    GPIOF->PUR |= SWITCH;                    // Enable pull-up resistor on PF4
    GPIOF->DIR |= RED_LED;                   // Set PF1 as an output pin and PF4 as
    an input pin
    GPIOF->DEN |= (RED_LED | SWITCH);        // Enable PF1 and PF4 as a digital GPIO
    pins
    while(1) {
        state = GPIOF->DATA & SWITCH;        // Read the state of the switch
        if (state == 0) {                    // If the switch is pressed (since it's
    Pull-up)
            GPIOF->DATA |= RED_LED;          // Turn on the LED
        } else {                             // If the switch is not pressed
            GPIOF->DATA &= ~RED_LED;         // Turn off the LED
        }
    }
}
```

Listing 5.7: Polling-based switch input reading

Explanation:

- `GPIOF->PUR |= SWITCH;` enables the internal pull-up resistor on PF4.
- The `while(1)` loop continuously reads the switch state.
- When `state == 0`, the button is pressed (pull-up makes it active-low).
- The LED is controlled directly in the main loop without interrupts.

5.2.1.2 Example 2 — Interrupt-Driven Switch Input

This example configures SW1 and SW2 to generate interrupts on button presses and toggles the green LED in the ISR.

```
#include "TM4C123.h"

#define GREEN_LED 0x08
#define SW1 0x10
#define SW2 0x01

int main(void)
{
    SYSCTL->RCGCGPIO |= (1<<5);              // Enable clock to GPIOF

    GPIOF->LOCK = 0x4C4F434B;                // unlock commit register
    GPIOF->CR = 0x01;                        // make PORTF0 configurable


    // PINS Configurations
    GPIOF->DIR &= ~(SW1 | SW2);              // Set SW1, SW2 as input pins
    GPIOF->DIR |= GREEN_LED;                 // Set GREEN_LED as output pin

    GPIOF->DEN |= (GREEN_LED | SW1 | SW2);  // Enable digital function for
    GREEN_LED, SW1, SW2
    GPIOF->PUR |= (SW1 | SW2);               // Enable pull-up resistors on SW1, SW2

    // Interrupt Configurations
    GPIOF->IS  &= ~(SW1 | SW2);              // SW1, SW2 are edge-sensitive
    GPIOF->IBE &= ~(SW1 | SW2);              // SW1, SW2 are not both edges
    GPIOF->IEV &= ~(SW1 | SW2);              // falling edge trigger
    GPIOF->ICR |= (SW1 | SW2);               // clear any prior interrupt
    GPIOF->IM  |= (SW1 | SW2);               // unmask interrupt

    // Enable Interrupts
    NVIC->ISER[0] |= (1<<30);                // enable interrupt 30 in NVIC
    // NVIC_EnableIRQ(GPIOF_IRQn);           // Alternative way using CMSIS function
    while(1)
    {
    }
}

void GPIOF_Handler(void) {
    if (GPIOF->MIS & SW1) {                  // check if interrupt causes by PF4/SW1
        GPIOF->DATA |= GREEN_LED;            // turn on green LED
        GPIOF->ICR |= SW1;                   // clear the interrupt flag
    }
    else if (GPIOF->MIS & SW2) {             // check if interrupt causes by PF0/SW2
        GPIOF->DATA &= ~GREEN_LED;           // turn off green LED
        GPIOF->ICR |= SW2;                   // clear the interrupt flag
    }
}
```

Listing 5.8: Interrupt-driven switch input handling

Explanation:

- `GPIOF->LOCK = 0x4C4F434B;` unlocks Port F for PF0 configuration.

- `GPIOF->CR = 0x01;` allows changes to PF0.
- `GPIOF->IS &= ~(SW1 | SW2);` configures edge-sensitive interrupts.
- `GPIOF->IEV &= ~(SW1 | SW2);` selects falling-edge trigger (button press).
- `GPIOF->ICR |= (SW1 | SW2);` clears any prior interrupt flags before enabling.
- `GPIOF->IM |= (SW1 | SW2);` unmasks (enables) interrupts for SW1 and SW2.
- `NVIC->ISER[0] |= (1«30);` enables GPIO Port F interrupt in NVIC.
- The ISR checks `GPIOF->MIS` to identify which switch caused the interrupt.
- Each interrupt flag must be cleared with `GPIOF->ICR` to prevent re-triggering.

5.2.2 Tasks

5.2.2.1 Task 1 — Toggle LED Using Polling

Modify Example 1 to toggle the green LED (PF3) with each press of SW1 (PF4). The LED should change state (ON→OFF or OFF→ON) every time the button is pressed and released.

Requirements:

- Use polling to detect button presses.
- Implement software debouncing by adding a 20 ms delay after detecting a press.
- Toggle the LED state instead of simply turning it on or off.
- Ensure the LED changes state only once per button press.

5.2.2.2 Task 2 — LED Sequence Using Interrupts

Modify Example 2 to cycle through LED colors using interrupts:

- Pressing SW1 (PF4) cycles through: **Red → Blue → Green → Red...**
- Pressing SW2 (PF0) cycles through: **Yellow → Magenta → Cyan → Yellow...**

Requirements:

- Use interrupt-driven input handling.
- Maintain separate state variables for each button's LED sequence.
- Update the LED color in the ISR based on the current state.
- Clear interrupt flags properly to avoid repeated triggering.

Hint: Use a global variable (e.g., `sw1_state`) to track the current position in the sequence. In the ISR, increment the state and use a switch-case or modulo operation to cycle through colors. Refer to Table 4.5 in Experiment 4 for LED color codes.