Task 2: Debounce Problem in Push Buttons

Abdo Wael

Objective

This task investigates a common issue encountered when reading input from mechanical push buttons — the debounce problem — and demonstrates how to overcome it using a software solution applied to the previous task's circuit.

Problem Statement

Mechanical push buttons do not produce a clean digital signal when pressed. Instead of transitioning cleanly from HIGH to LOW (or vice versa), they tend to "bounce" due to their physical contact mechanics. This results in multiple false triggers during a single press.

Problem name: Button Bouncing (Switch Bouncing or Contact Bounce)

Observed Symptoms

When the bounce occurs:

- The Arduino may detect multiple presses instead of one.
- LEDs may flicker or toggle unexpectedly.
- The behavior becomes inconsistent, especially in logic-sensitive circuits.

Solutions

There are two ways to solve the bouncing issue:

1. **Hardware Debouncing:**
    - Add a capacitor across the button.
    - Use Schmitt triggers or RC filters.
2. **Software Debouncing:**
    - Implement a delay or time-based filter in code to ignore rapid transitions.

Our Approach

In this task, we use a **software debouncing solution** using a timer (based on `millis()`) to filter out bouncing signals. The circuit is the same as Task 1 with one LED and one push button.

Tinkercad Link

Click here to view the project in Tinkercad

Circuit Diagram

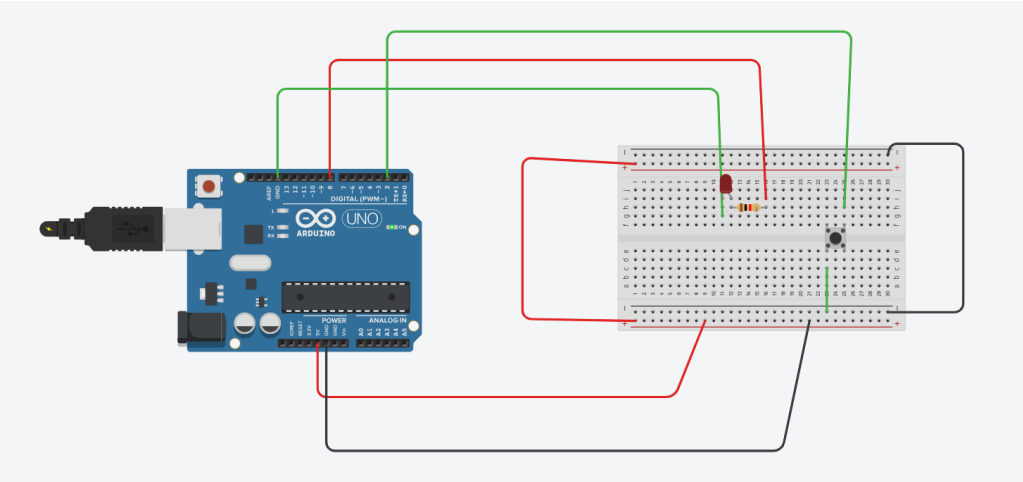

Figure 1: Circuit diagram with debounce logic implementation

Arduino Code with Debounce

```
const int pushButtonPin = 2;
const int ledPin = 8;

bool ledState = LOW;
bool lastButtonState = HIGH; // previous button state
unsigned long lastDebounceTime = 0;
unsigned long debounceDelay = 50;

void setup() {
    pinMode(pushButtonPin, INPUT_PULLUP); // using internal pull-up
    pinMode(ledPin, OUTPUT);
    digitalWrite(ledPin, ledState);
    Serial.begin(9600);
}

void loop() {
    int reading = digitalRead(pushButtonPin);

    // check if button state changed
    if (reading != lastButtonState) {
        lastDebounceTime = millis(); // reset debounce timer
    }

    if ((millis() - lastDebounceTime) > debounceDelay) {
```

```
        // stable state detected
        if (reading == LOW && lastButtonState == HIGH) {
            // button just pressed
            ledState = !ledState; // toggle LED
            digitalWrite(ledPin, ledState);
            Serial.println(ledState ? "LED␣ON" : "LED␣OFF");
        }
    }

    lastButtonState = reading;
}
```

How the Code Works

- It continuously monitors the button's state.
- If a change is detected, a timer is started using `millis()`.
- Only if the state remains stable for more than 50 milliseconds is it considered a valid press.
- The LED toggles only after a stable press is detected.

Advantages of This Method

- Eliminates false button presses.
- Reliable response in real-time applications.
- Does not require external components (pure software).

Enhancement Ideas

- Combine software and hardware debounce for better reliability.
- Abstract the debounce code into a reusable function or class.
- Add long-press detection or multiple-button support.

Conclusion

The debounce problem is a common issue in embedded systems. Using the debounce technique described above, we can significantly improve input reliability in projects like the LED toggle circuit. This method is efficient, resource-friendly, and scalable.