LE/ESSE 2220 Algorithmic and Computational Methods

Lab 2: Software Debouncing and Design Your Own Space Battery Display

(Fall 2025-2026)

Z. ARJMANDI (ZARARJ@YORKU.CA)

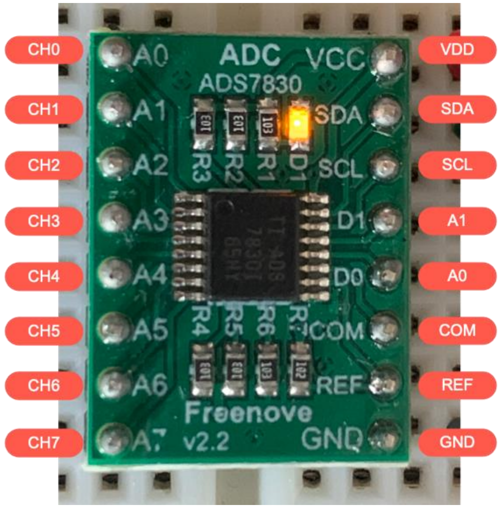
YORK U

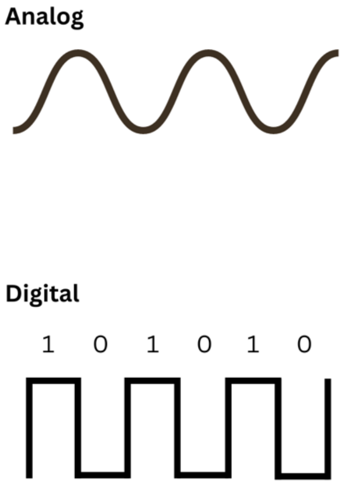
Review

Raspberry Pi GPIO – Inputs & Outputs

- **Inputs (read, sense)**
  - **Digital Input (0 / 1, LOW / HIGH)**
    - Button, switch, motion sensor
    - Data type: **Boolean (True/False)**
  - **Analog Input** (needs external ADC)
    - Temperature sensor, potentiometer
    - Data type: **Integer/Float** (e.g., 0–1023)

- **Outputs (write, interact)**
  - **Digital Output (0 / 1, OFF / ON)**
    - LED, buzzer, relay
    - Data type: **Boolean (True/False)**
  - **PWM Output (Pulse Width Modulation)**
    - LED dimming, motor speed control, servo angle
    - Data type: **Duty Cycle % (0–100)**

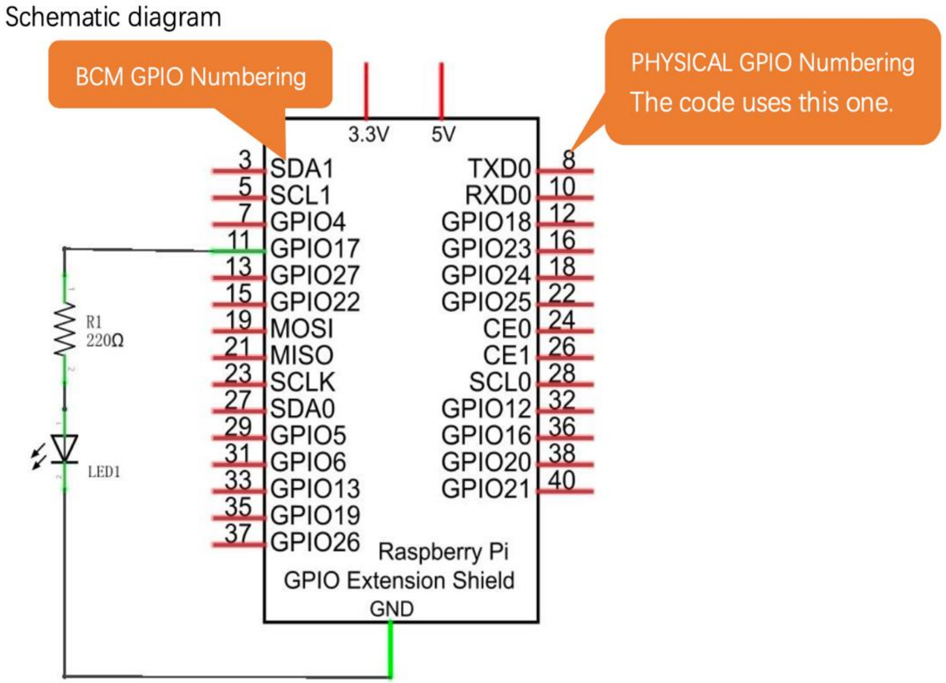




Raspberry Pi GPIO: BCM vs Physical

- **GPIO (General Purpose Input/Output):**
  - Pins on the Raspberry Pi used to control devices (LEDs, motors) or read data from sensors.
  - Can be configured as **input** or **output** in Python.

- Numbering Systems
  - **BCM GPIO (Broadcom SOC Channel):**
    - Refers to the *chip's internal numbering scheme*.
    - Example: **GPIO17**.
  - **Physical GPIO (Board Numbering):**
    - Refers to the *pin's actual position* on the 40-pin header.
    - Example: **Pin 11**.



Raspberry Pi GPIO: BCM vs Physical

- Here’s the equivalent code side-by-side for **BCM** vs **BOARD** numbering. Both turn an LED on and off on the same physical pin (pin 11 on the header, which is GPIO 17 in BCM mode):

```python
import RPi.GPIO as GPIO

GPIO.setmode(GPIO.BCM)        # Use BCM numbering
GPIO.setup(17, GPIO.OUT)      # GPIO 17 = physical pin 11

GPIO.output(17, GPIO.HIGH)    # LED ON
GPIO.output(17, GPIO.LOW)     # LED OFF

GPIO.cleanup()                # Reset pins
```

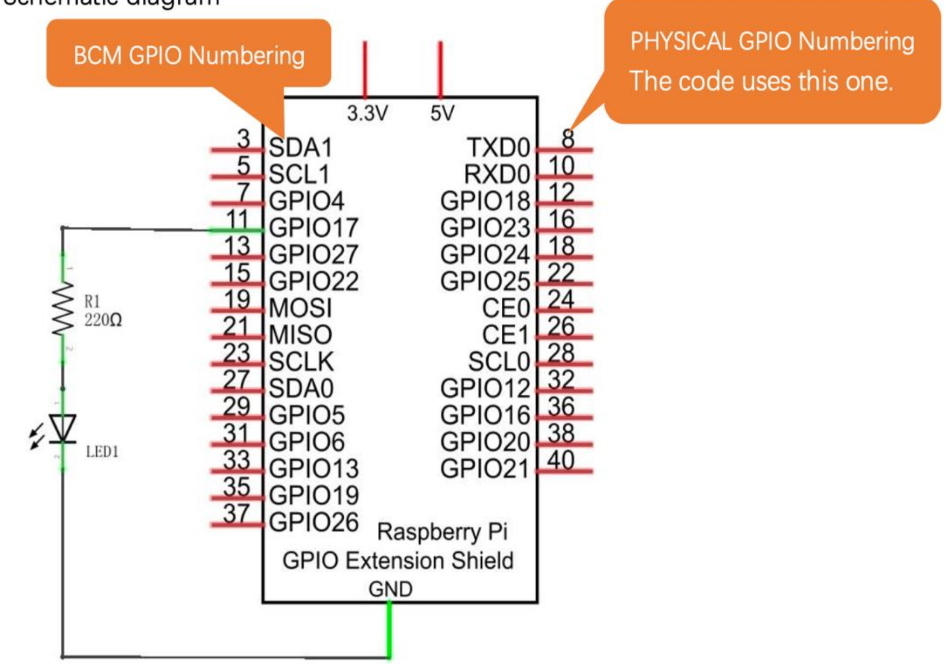
```python
import RPi.GPIO as GPIO

GPIO.setmode(GPIO.BOARD)     # Use physical pin numbers
GPIO.setup(11, GPIO.OUT)     # Pin 11 = GPIO 17 (same pin as above)

GPIO.output(11, GPIO.HIGH)   # LED ON
GPIO.output(11, GPIO.LOW)    # LED OFF

GPIO.cleanup()               # Reset pins
```

Schematic diagram



Using GPIO on Raspberry Pi

- **GPIO.setmode()** → choose numbering system (**BCM** or **BOARD**)
- **GPIO.setup(pin, GPIO.OUT)** → set a pin as **output** (to send signals, e.g. turn on an LED)
- **GPIO.setup(pin, GPIO.IN)** → set a pin as **input** (to read signals, e.g. detect a button)

Example: Output (LED)

```
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)          # Use BCM numbers
GPIO.setup(17, GPIO.OUT)        # Pin 17 = Output

GPIO.output(17, GPIO.HIGH)      # LED ON
GPIO.output(17, GPIO.LOW)       # LED OFF
GPIO.cleanup()                  # Reset pins
```

Example: Input (Button)

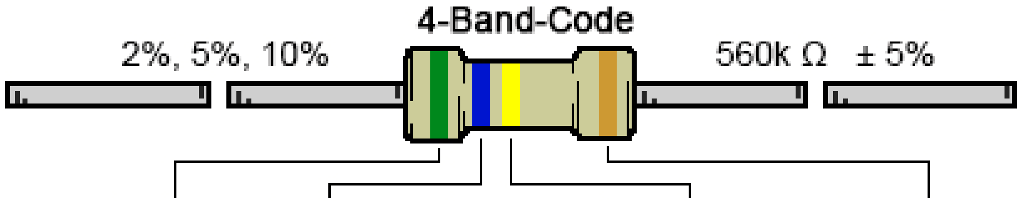
```
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)          # Use BCM numbers
GPIO.setup(18, GPIO.IN)         # Pin 18 = Input

if GPIO.input(18) == GPIO.HIGH:
    print("Button pressed")
else:
    print("Button not pressed")
GPIO.cleanup()
```

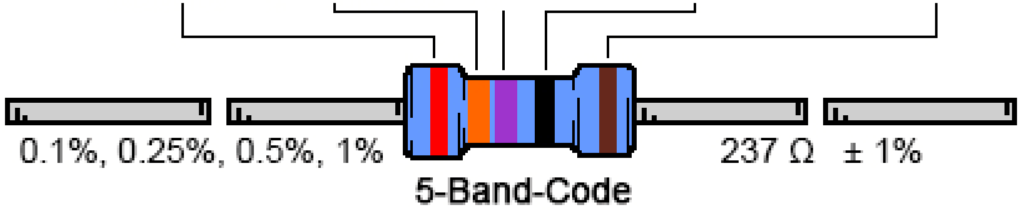
Resistor Color Code Chart

- 4-band:
  - Band 1 = First digit
  - Band 2 = Second digit
  - Band 3 = **Multiplier**
  - Band 4 = Tolerance

- 5-band:
  - Band 1 = First digit
  - Band 2 = Second digit
  - Band 3 = Third digit
  - Band 4 = **Multiplier**
  - Band 5 = Tolerance



COLOR	1ST BAND	2ND BAND	3RD BAND	MULTIPLIER	TOLERANCE
Black	0	0	0	1Ω	
Brown	1	1	1	10Ω	± 1% (F)
Red	2	2	2	100Ω	± 2% (G)
Orange	3	3	3	1KΩ	
Yellow	4	4	4	10KΩ	
Green	5	5	5	100KΩ	± 0.5% (D)
Blue	6	6	6	1MΩ	± 0.25% (C)
Violet	7	7	7	10MΩ	± 0.10% (B)
Grey	8	8	8	100MΩ	± 0.05%
White	9	9	9	1GΩ	
Gold				0.1Ω	± 5% (J)
Silver				0.01Ω	± 10% (K)



Review: Modules and Imports Basics

- A **module** is just a Python file (.py) that contains code (functions, classes, variables).
- You can **import** that file into another Python file to reuse the code.
- This helps keep code organized and avoids repeating yourself.

- **Example: Creating a Module**

math_tools.py

```python
# This is our module file

def add(a, b):
    return a + b

def subtract(a, b):
    return a - b
```

- **Importing the Module**

main1.py

```python
# Import the whole module
import math_tools

print(math_tools.add(5, 3))        # 8
print(math_tools.subtract(10, 4)) # 6
```

main2.py

```python
# Import specific functions
from math_tools import add

print(add(2, 3))   # 5

# Import with an alias (nickname)
import math_tools as mt

print(mt.add(1, 1))   # 2
```

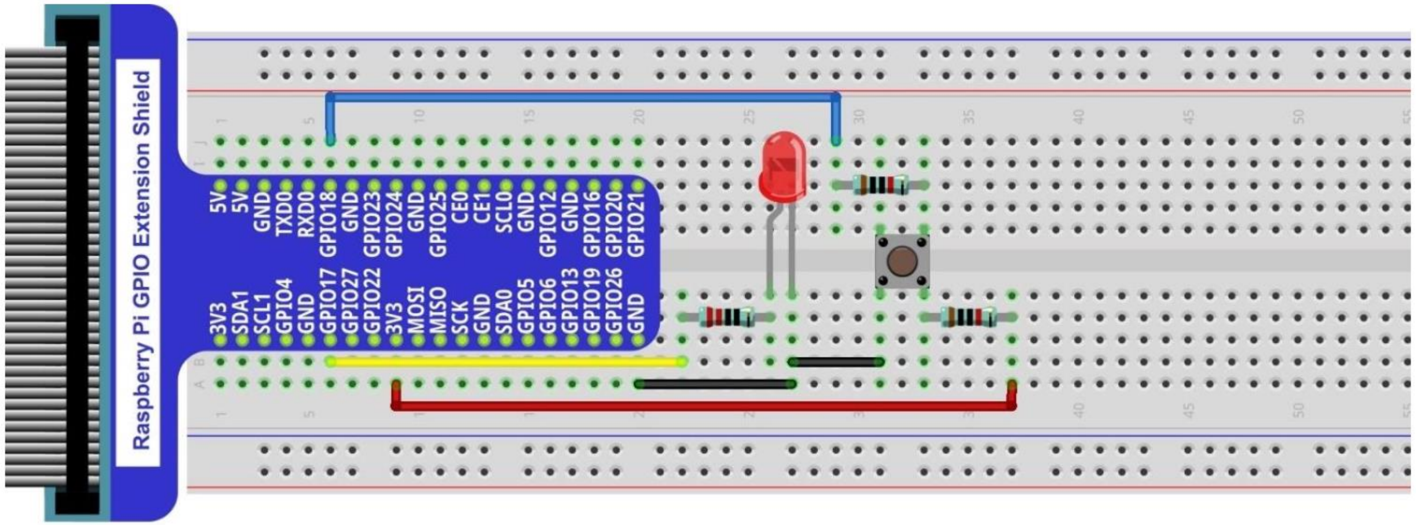
Lab 2

Lab 2

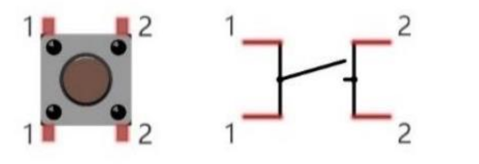
- **Part 1:** Software Debouncing with GPIO
  - Buttons & LEDs
- **Part2:** Design Your Own Space Battery Display
  - LED Bar Graph

Lab 2 Part 1: Software Debouncing with GPIO

Buttons & LEDs



Inputs: Push Button “Bounce” & Debouncing



What happens:

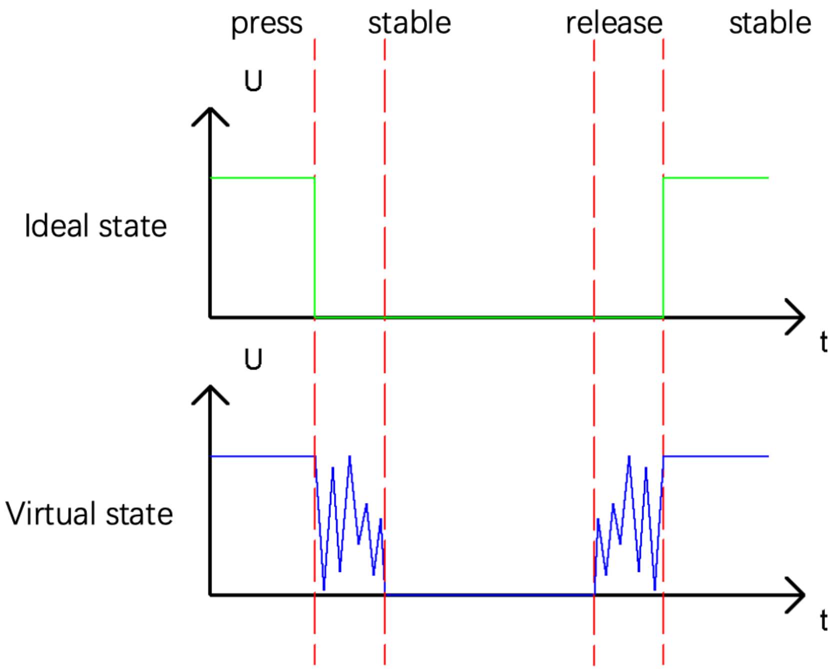
- When a push button is pressed or released, tiny **mechanical vibrations** occur.
- Instead of switching **cleanly ON or OFF**, the signal rapidly fluctuates between HIGH and LOW.
- This happens in **milliseconds** → too fast for humans but detected by microcontrollers.

Problem:

- One press can be misread as **multiple presses/releases**.
- Causes false triggers or unreliable behavior in programs.

Solution (Debouncing):

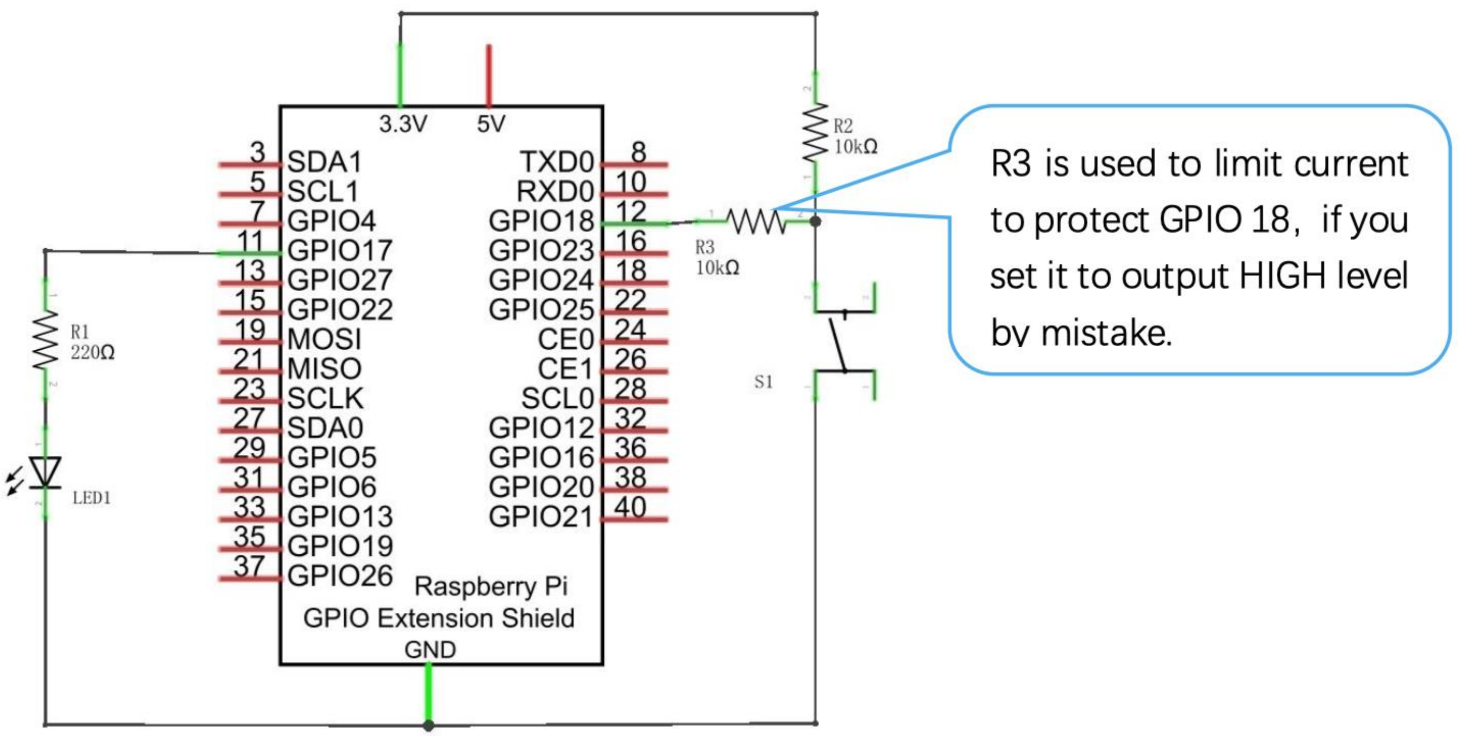
- Check the button state **multiple times** over a short delay.
- Only accept the press if the state is **stable (unchanged)**.
- Ensures a single, clean ON/OFF detection.



Lab 2 Part 1: Software Debouncing with GPIO

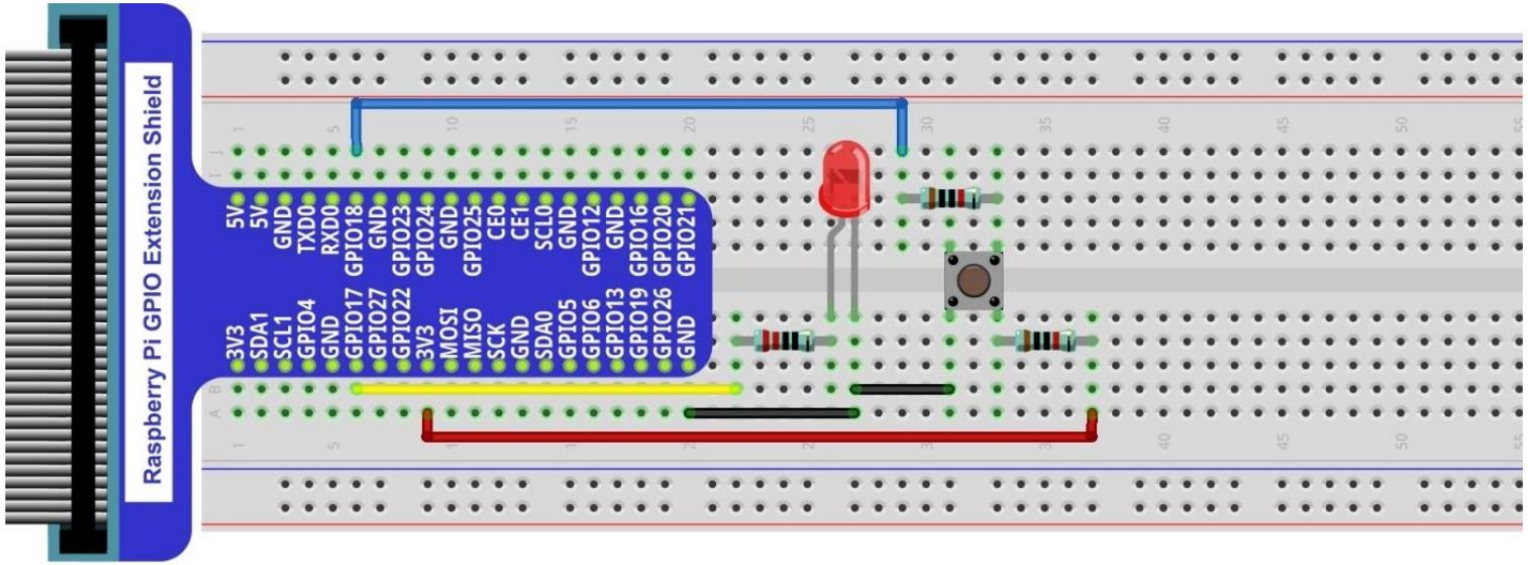
- For more details on this circuit, please refer to Tutorial Chapter 2 Buttons & LEDs (available on E-Class).

Schematic diagram



Lab 2 Part 1: Software Debouncing with GPIO

- Build the Circuit



Lab 2 Part 1: Software Debouncing with GPIO

Step 1 – Build the Circuit

- Connect the components to the Raspberry Pi following the provided circuit diagram.

Step 2 – Run the Sample Program

- Execute Tablelamp.py to verify that both your code and circuit are working correctly.

Step 3 – Test Without Debouncing

- Remove the bouncetime argument from the code.
- Press the button once and observe what happens.
- Do you notice multiple toggles from a single press?

Step 4 – Add Software Debouncing

- Reintroduce the bouncetime argument and test different values: **50, 100, 200, 300, 500, 1000**.
- Press the button normally and then quickly several times.
- Record how many presses are detected for each setting.

Step 5 – Record Your Observations

- Fill in the table below with your results:

bouncetime (ms)	What happens when I press once?	What happens when I press quickly?
0 (none)	???	???
50	???	???
…	???	???

Understanding add_event_detect()

- **buttonPin** → the pin connected to the button.
- **GPIO.FALLING** → detect when the signal goes from HIGH to LOW (button press).
- **callback=buttonEvent** → run the function buttonEvent() when the button is pressed.
- **bouncetime=300** → ignore extra signals for 300 ms to prevent multiple triggers from one press.

```
GPIO.add_event_detect(buttonPin, GPIO.FALLING, callback=buttonEvent, bouncetime=300)
```

Part 1 Report Format (short, personal, verifiable)

Setup

- Attach a clear photo of your circuit.
- Copy and paste your final code.

Output

- Include a screenshot of your program running (showing the LED toggling and print messages).

Observations

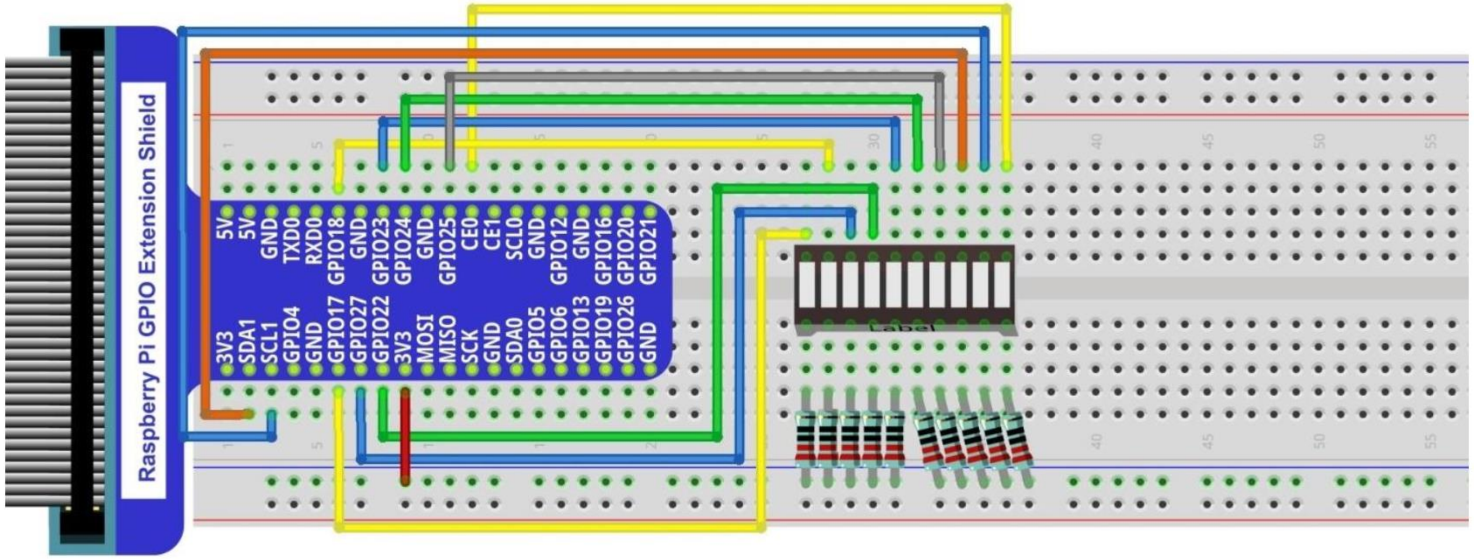
- What happened when you removed the bouncetime argument?
- Fill in your results table for different debounce times (50, 100, 200, 300, 500, 1000 ms).

Analysis

- Which debounce value worked best for your button? Why?
- In your own words, what is “button bounce” and why do we need debouncing?

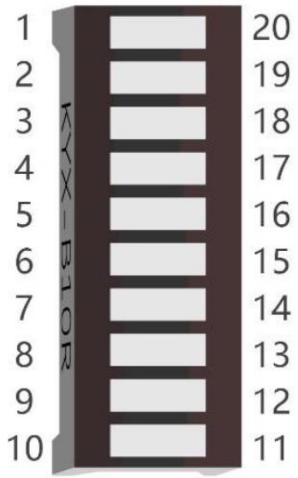
Lab 2 Part 2: Design Your Own Space Battery Display

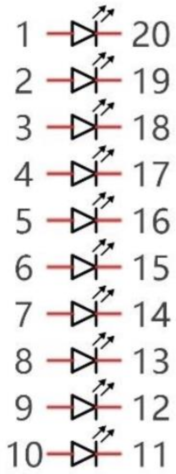
LED Bar Graph

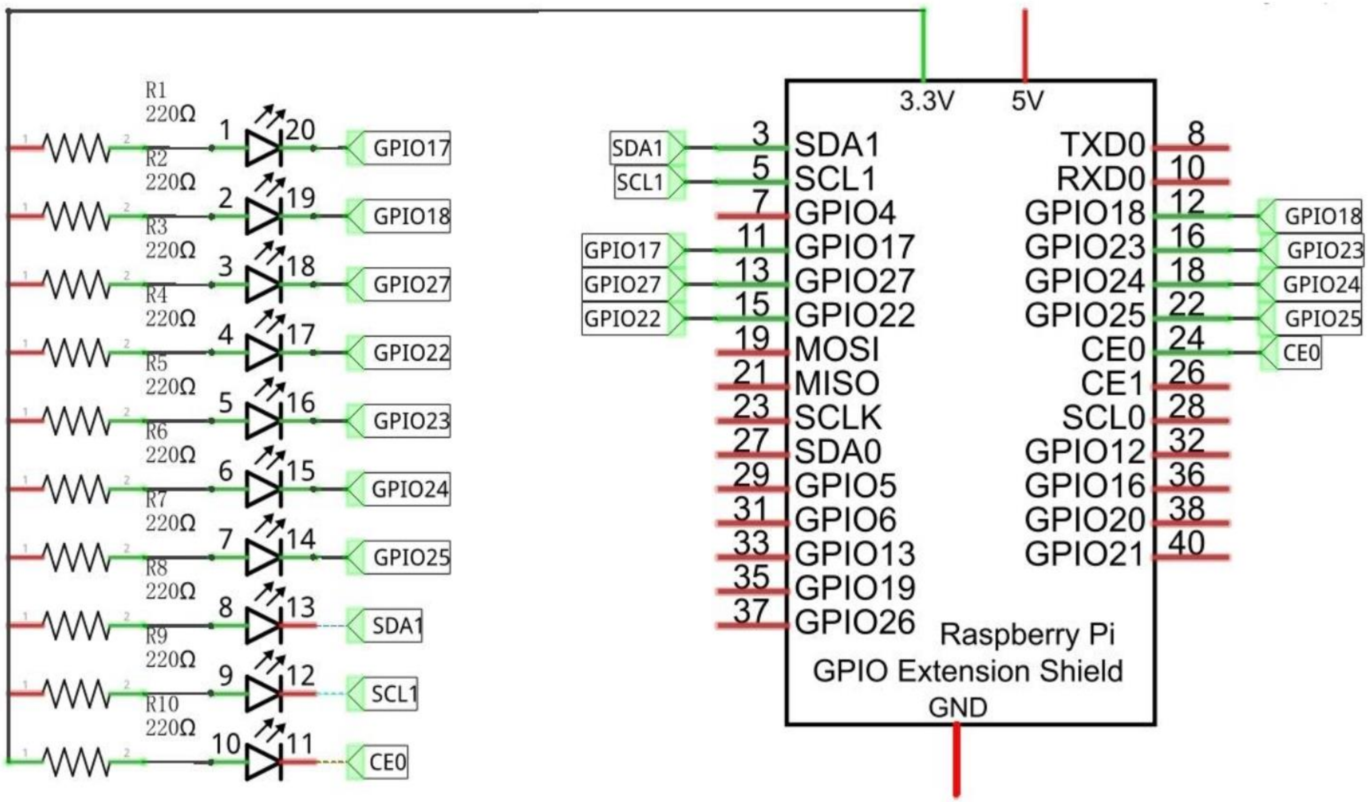


Lab 2 Part 2: Design Your Own Space Battery Display

- A Bar Graph LED has 10 LEDs integrated into one compact component.
- For more details on this circuit, please refer to Tutorial Chapter 3 LED Bar Graph (available on E-Class).







Lab 2 Part 2: Design Your Own Space Battery Display

- Show a “spaceship battery” on the 10-LED bar.
- The battery percentage controls **how many LEDs should be on**.
- You will write a new file space_controller.py that decides *what to light* and *how to light it*. You can do this **with a tiny class** or **with simple functions,** your choice.

Lab 2 Part 2: Design Your Own Space Battery Display

Requirements (what we'll check)

- Create a new file **space_controller.py**.
- In your main loop, call a function (or class method) from space_controller.py.
- That function (or method) will:
  - **Receive a battery percentage** (0–100).
  - **Decide how many LEDs should be ON** based on that percentage.
  - **Decide how to light them up** (your choice: one-by-one, all at once, blinking, etc.).
- The number of LEDs that are lit must always **match the battery %** you are targeting.
- Your chosen lighting style must be **clearly visible** when you run the program.

Bonus: suggested mini-challenges (pick any)

- Low-battery warning (<15%): blink the first LED three times before settling.
- Charging effect when going up: blink the "next" LED once before turning it on.
- All-at-once jump: instantly match the new level, then do a brief sparkle.
- Smooth mode: use shorter delays for larger jumps (fast when far, slow when close).

Hints to Get Started (Optional)

- **Mapping Percentage to LEDs** – how to turn 0–100% into 0–10 LEDs.
- **Tracking Current vs. Target LEDs** – do you need to remember what's already lit?
- **Lighting Styles** – one-by-one, all-at-once, blinking, or other creative effects.
- **Delays and Timing** – how to control the speed of changes.
- **On vs. Off Logic** – decide when to switch an LED on or turn it off again.

Part 2 Report Format (short, personal, verifiable)

- **Setup**
  - Attach a clear photo of your circuit (showing the LED bar connected).
  - Copy and paste your final code (both LightWater.py and space_controller.py).
  - Include meaningful comments in your code to explain what each part does.
- **Demo**
  - Ask your TA or instructor to come and check your code running on the hardware.
- **Observations**
  - Explain the lighting style you designed and its logic (one-by-one, all-at-once, blinking, etc.).
  - Describe how the number of LEDs changes as the battery % changes.
- **Analysis**
  - Explain each of your functions or methods in your own words (what it receives, what it does, and what it returns
  - If you used a class, what advantage did it give you? If you used functions, why did you prefer that approach?
- **Bonus (Optional)**
  - If you attempted a bonus challenge (low battery warning, charging effect, sparkle, etc.), describe what you implemented.

Rubric

- **Part 1: Button & Debounce (4 pts)**
  - **Circuit setup** (photo of wiring, pins explained) – **1 pt**
  - **Code** (runs correctly + comments included) – **1 pt**
  - **Observations** (bouncetime removed/tested, results table filled) – **1 pt**
  - **Analysis** (best debounce value chosen + explanation of button bounce) – **1 pt**
- **Part 2: Space Battery Indicator (6 pts)**
  - **Circuit setup** (photo of LED bar wiring, pins explained) – **1 pt**
  - **Code and Demo** (runs correctly + comments included) – **3 pts**
  - **Design & Analysis** (LED count matches battery %, chosen lighting style explained, functions/methods described) – **2 pts**
- **Bonus (optional, up to +2 pt)**
  - Extra effect implemented (low-battery blink, charging animation, sparkle, etc.) – **+2 pt**
- **Due Date:**
  - Monday, 11:59 PM
- **Quiz 2**
  - Release: Monday, 11:59 PM
  - Deadline: Friday, 12:00 PM (before labs)

Next Week

- Lab 03:
  - Part 1: Buzzer (Chapter 6)
  - Part2: Joystick (Chapter 12)