LESSON 5

SWITCHES AND CORRECTING FOR SWITCH BOUNCE

OBJECTIVE

In this lesson, you will learn to work with different types of switches and learn to use programs to overcome the electrical shortcomings of some switches.

MATERIALS

- Raspberry Pi connected to a monitor, keyboard, and mouse
- Circuit from Lesson B-4
- 1 x Slide Switch
- 1 x Pushbutton Switch
- 3 x Long Jumper Wires
- 2 x Short Jumper Wires

REVIEW CONCEPTS

If you do not feel comfortable with the following concepts, please review them before proceeding.

- Working with switches (Lesson A-5)
- Boolean Logic, If/Else Statements (Lesson A-14)
- GPIO Pin States, GPIO Pin Cleanup (Lesson A-16)

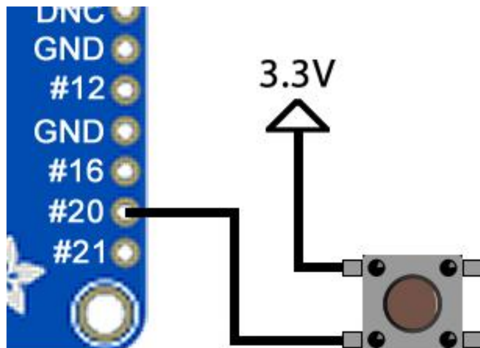
LESSON

In this lesson, you will learn to work with different types of switches including both slide and pushbutton switches to accomplish various tasks. You will also learn to write code to overcome an electrical shortcoming of switches known as switch bounce.

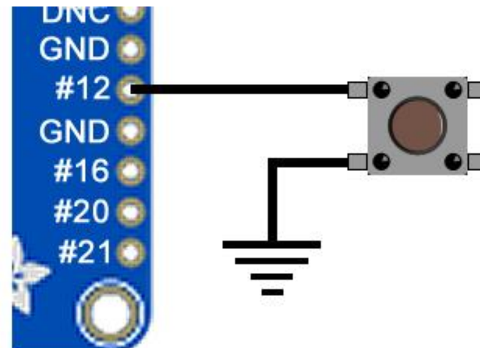
PULL-UP / PULL-DOWN OPTIONS

In Level A, you learned how to use resistors in a specific configuration to keep inputs reading high or low, when the pushbutton switch was not pressed. This was opposed to inputs being left floating, where an input is not connected to anything until a switch is pressed, which can lead to inputs being read unreliably by your program.

While it's good to understand the basics behind using resistors for pull-up and pull-down, the Raspberry Pi can help you simplify circuit design by doing the pull-up and pull-down work for you. The Raspberry Pi contains internal resistors that can be accessed when setting up a GPIO pin as an input. This will simplify your switch wiring to a single jumper wire from the GPIO pin to the switch, and another wire from the switch to 3.3V or ground, depending on whether you want a high or low to trigger the input.



Active high - needs pull down in software



Active low - needs pull up in software

In Python, pulling an input up or down can be done when the pin is assigned as an input, by supplying a third parameter called **pull_up_down**:

```
GPIO.setup(12, GPIO.IN, pull_up_down=GPIO.PUD_UP)
```

This command will initialize **GPIO12** as an input, and configure that input to be internally pulled up. This input will be an active low, which means this input will remain high until it is made active by connecting it to ground or low, hence the name "active low". **GPIO12** will now only need to be connected to ground to trigger the input, which can be done very simply using a switch and two wires.

The opposite of active low is active high, which means the input will remain grounded or low until it is made active by connecting to 3.3V. This can be accomplished by using **GPIO.PUD_DOWN** after the **pull_up_down** parameter:

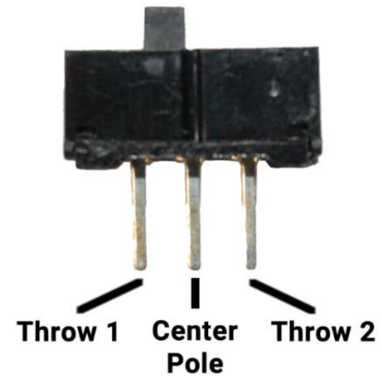
```
GPIO.setup(20, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
```

This command will initialize **GPIO20** as an input and configure that input to be internally pulled down. **GPIO20** will now only need to be connected to 3.3V to trigger the input, which can be done with a switch and two wires.

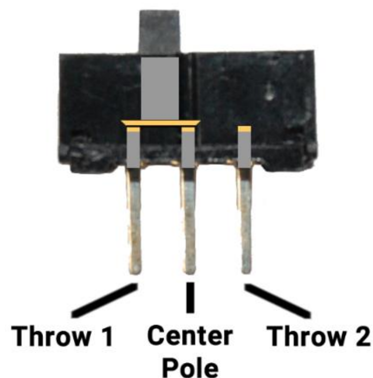
SLIDE OR TOGGLE SWITCH

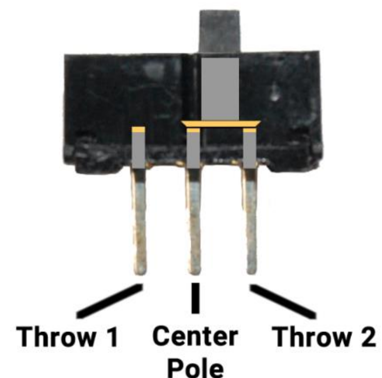
Until now, you have only worked with pushbutton switches that make connection when they are pressed, but lose that connection when they are released. There may come a time when you want a switch to stay connected without being pressed. This type of switch is called a toggle switch.

A toggle switch "toggles" one or more inputs between one or more outputs. One very common type of toggle switch is a two-position slide switch:



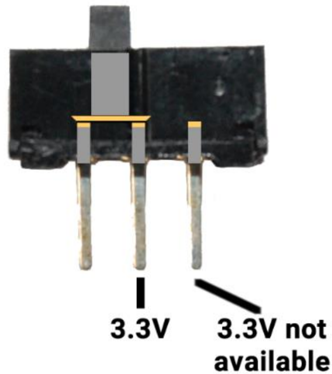
In a two-position slide switch the center pole can connect to one of the throws at a time, based on the position of the slider:



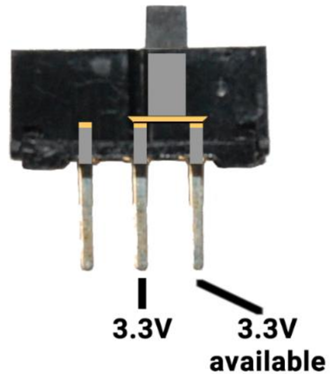


These switches can be used as power on/off switches for electronic devices by connecting power through the center pole as well as one of the throws:

Slide switch used for power control in OFF position

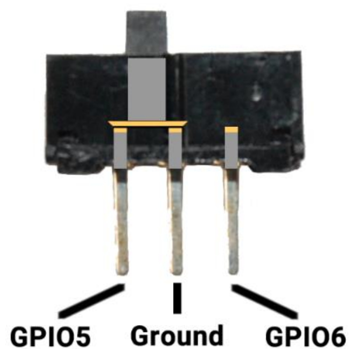


Slide switch used for power control in ON position

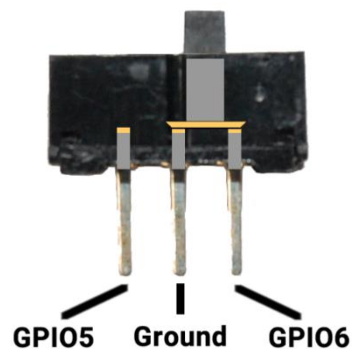


A slide switch can also be used as a selector between two different inputs. In the diagram below, a slide switch is connected to two GPIO pins, with the center terminal attached to ground.

Slide switch connecting ground to GPIO5



Slide switch connecting ground to GPIO6



In one position, GPIO5 is grounded and a GPIO6 will be internally pulled up by the Raspberry Pi. In the other position, GPIO6 will be grounded and GPIO5 will be internally pulled up by the Raspberry Pi.

Programming for a slide switch is identical to the programming you've already done for pushbutton switches. Your program could use these switch inputs to light up different LEDs, modify the frequency of a piezo buzzer, or anything else you might want to control.

SWITCH BOUNCE

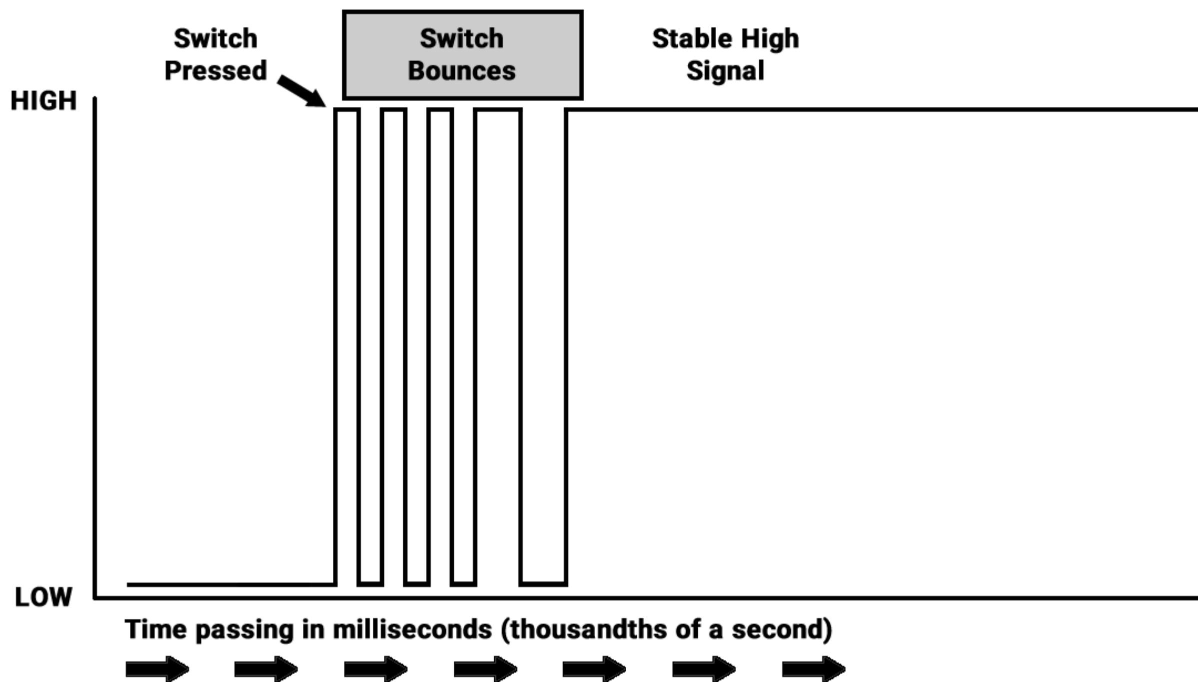
When switches are opening or closing, metal plates called contacts must come together or move apart to control the flow of electricity through the switch. These plates are made of metal, so they will conduct electricity, but this metal-to-metal contact can result in the two plates bouncing off each other a very tiny amount, until they settle either fully together, or fully apart. This microscopic movement can result in the switch being open and closed very quickly, typically less than one millisecond, until the switch settles into the desired state. This phenomenon is referred to as switch bounce.

All mechanical switches will exhibit some level of switch bounce, but it may not be a problem in their specific application. For example, the light switch in your room has switch contact bounce but you don't see the light flash on and off multiple times when you flip the switch on. This is due to the latency in overhead lighting and most other electronics.

Switch bounce may become a problem depending on how you plan to use the switch input. If the program just turns on an LED based on switch input, then bounce won't really be a problem. Extremely rapid on/off LED activity will be masked by the amount of time it takes the LED to turn on and off, so the LED will appear to turn on and off smoothly, even though switch bounce is still occurring.

The problem occurs when you have the ability to check a switch very quickly, like the Raspberry Pi is able to do, thousands of times per second. Imagine you have program that is supposed to keep track of how often you drink water each day. Each time you drink a glass of water you press a pushbutton, and the program keeps track of how many times the switch went from low to high, adding to that count throughout the day.

Pressing the button quickly, only one time, should result in one being added to the current count. Due to switch bounce, there may be a few extra low to high transitions for every time the button is pushed, and these will all be logged to the counter.



Pressing the button 8 times throughout the day might result in a final count of 32 glasses of water, which is obviously very inaccurate. Removing these extra counts is called debouncing, which can be done via hardware using additional components, or via software. In the interest of keeping electronic connections to the Pi as simple as possible, we will focus on debouncing using software in Python. You may find many effective methods for software debouncing in programs online. Below, we will detail a few of the simplest options.

One method of debouncing involves checking the input a certain amount of time after the first time an input is triggered. This will allow time for the bouncing to settle and the input state can be confirmed once it is steady. This delay is usually not very long, as a delay of around 0.02 seconds should be enough to allow the input to settle into either a steady high or low state. If GPIO6 was configured as an active low input, that code would look something like this:

```
while True:

    if GPIO.input(6) == False:

        time.sleep(0.02)

        if GPIO.input(6) == False:

            water_count = water_count + 1
```

The first if statement will trigger the very first time GPIO6 goes low. The program will sleep for 0.05 seconds, and then check GPIO6 again. If GPIO6 is still False, then one will be added to **water_count**. If this second check of GPIO6 is True then nothing else will happen, and the value of **water_count** will not be changed.

There are a couple of problems with this method. If you somehow managed to press the button for less than .02 seconds, no water would be recorded, as the second level if statement would not evaluate as False. Also, since this entire loop will be evaluated every .02 seconds, you can easily end up with multiple **water_count** increments for a single button press, which defeats the purpose of the delay.

You could increase the amount of delay to avoid multiple **water_count** events being counted when the button is held down. Increasing this delay will however increase the amount of time between the first and second checks, which could allow you a greater chance of pressing the button and releasing without being counted. You could tune this value to find a balance between quick presses not registering and longer presses registering more than once, but it may still not give you performance you would like.

Another method of debouncing includes adding a delay to the loop that runs after an input is triggered:

```
while True:
    if GPIO.input(6) == False:
        water_count = water_count + 1
        time.sleep(.1)
```

The very first time GPIO6 goes low the loop will be triggered. One will be added to **water_count** and the program will sleep for .1 seconds before resuming further checks. This will eliminate the possibility of quick button presses not being counted, and the delay in this loop can be modified to eliminate single button presses registering more than one count per press, without affecting the responsiveness of the first button press.

The only problem left with this code is that button presses longer than .1 seconds will continue to register additional **water_count** events. This can be fixed by adding a while loop to hold the program as long as GPIO6 remains low:

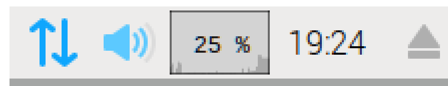
```
while True:
    if GPIO.input(6) == False:
        water_count = water_count + 1
        time.sleep(.1)
        while GPIO.input(6) == False:
            pass
        time.sleep(0.02)
```

The **pass** command in the new while loop tells that loop to do nothing when triggered, which is exactly what we want. As long as GPIO6 remains low the program will remain stuck in this while loop, doing nothing, and not adding to the **water_count** value. As soon as GPIO6 is no longer False, the rest of the initial if statement will run. The final **time.sleep(0.02)** was added to debounce the opening of the switch, as without the delay, the bouncing during release of the switch might cause **water_count** to be increased unintentionally.

USING A DELAY TO SAVE SYSTEM RESOURCES

Due to its tiny size, your Raspberry Pi has a limited amount of processing power that can be devoted to tasks. This processing power is how much work can be done by the CPU, which stands for Central Processing Unit. CPU utilization is a value that indicates how hard the CPU is working. A utilization value of 100% indicates that the CPU cannot handle any more work, and system performance will be highly degraded. A low CPU utilization value like 2% means that 98% of the CPU is still available and ready to do processing work.

The current utilization of the CPU is displayed in the top-right corner of the menu bar:



In this image, the CPU utilization is at 25%, meaning 25% of the available processing power is being used, with 75% still free.

If CPU utilization is too high, no processing power is left over for routine operating system tasks. By running a loop in your program with no delay, you can easily consume more system resources than you really need, causing instability in the Raspbian Operating System.

If you have a loop checking an input in a program, adding a small delay like 0.1 seconds will free up valuable resources, allowing the OS to complete all the tasks it needs in the background. Your program will likely run no different, however you will see a big difference in the CPU utilization number, as well as how much heat is being generated by your Raspberry Pi.

ACTIVITIES

In the following activities you will practice the input switch handling techniques from this lesson. Slide and pushbutton switches will be added to the circuit from Lesson #B-4 to allow for the programming of additional functionality.

ACTIVITY #1 – ADDING SWITCHES TO THE CIRCUIT

In this activity you will add switches to the circuit you built in Lesson B-4

STEP #1

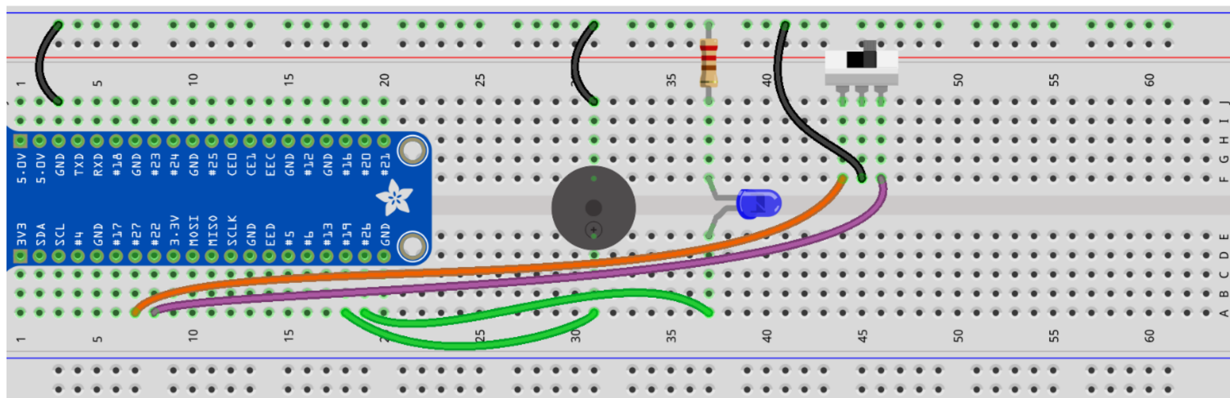
Make sure your Raspberry Pi is powered off. Using the circuit from Lesson B-4 as a starting point, add a slide switch connected between these points:

Slide switch in J44 through J46

Long jumper wire from F44 to A7

Short jumper wire from F45 to N2-41

Long jumper wire from F46 to A8



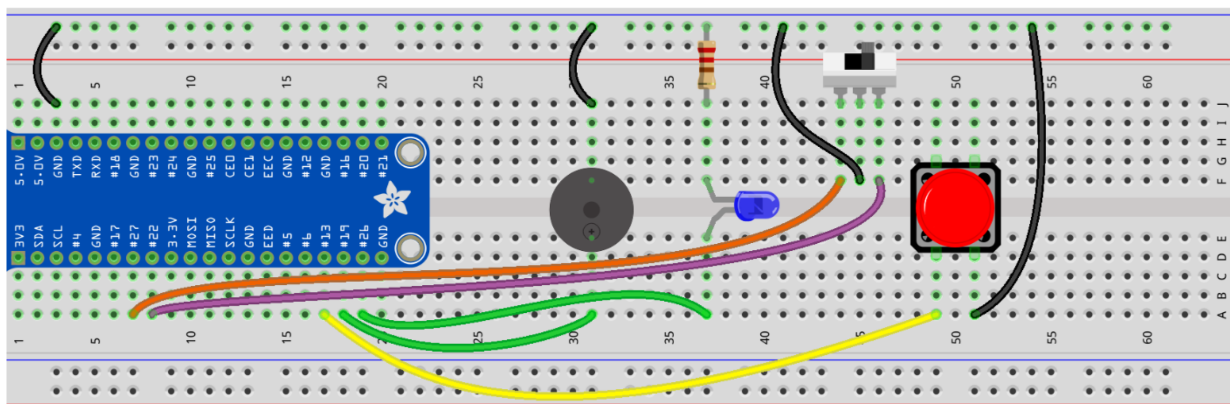
STEP #2

Next, you will add a pushbutton switch that will be used as a stop button for your program. Add a pushbutton switch connected between GPIO13 and ground.

Pushbutton switch in D49, D51, G49, and G51

Long jumper wire from A49 to A17

Short jumper wire from A51 to N2-54



The breadboard circuit is now ready for you to create a program that will use the switches to control some program functions.

ACTIVITY #2 – COUNTING BUTTON PRESSES

In this activity, you will create a program that will count each time the pushbutton connected to GPIO13 is pressed and print the current count to the console.

STEP #1

The first step will be to create an empty program that you can use to make a program that counts button presses. Create a new program in Thonny and save it to your Desktop as **button_counter**.

STEP #2

Since this program will interact with GPIO pins and have a delay, you must import the **RPi.GPIO** and **time** modules. Import these modules at the top of your program:

```
import RPi.GPIO as GPIO, time
```

STEP #3

Next, the GPIO pin mode will need to be set to BCM. **GPIO13** will also need to be configured as an active-low input since it's connected to ground. Since this switch does not have any pull-up resistors attached, this will need to be configured within the program. Add the following two highlighted lines just below your import code:

```
import RPi.GPIO as GPIO, time

GPIO.setmode(GPIO.BCM)

GPIO.setup(13, GPIO.IN, pull_up_down=GPIO.PUD_UP)
```

STEP #4

A variable will need to be used to hold the amount of button presses that have occurred, and this value should be zero when the program starts. Create a variable called **`button_count`** and set it equal to zero:

```
import RPi.GPIO as GPIO, time

GPIO.setmode(GPIO.BCM)

GPIO.setup(13, GPIO.IN, pull_up_down=GPIO.PUD_UP)


button_count = 0
```

STEP #5

Since this program is interacting with GPIO pins, use a **`try/except`** loop so the **`except:`** condition can catch keyboard interrupts and run a **`GPIO.cleanup()`** before the program exits. Add the following code to the bottom of your program:

```
button_count = 0


try:


except KeyboardInterrupt:

    GPIO.cleanup()
```

Your program can now exit smoothly, and without any errors, when the stop button is pressed in Thonny.

STEP #6

Now it's time to build the main loop that will watch **GPIO13** for a low or **False** state. If **GPIO13** is low, the button has been pressed, and the value of **button_count** will be incremented by 1. You will also use this loop to print the current value of **button_count**. Add the highlighted block of code to the end or your program:

```
button_count = 0


try:
    while True:
        if GPIO.input(13) == False:
            button_count = button_count + 1
            print(button_count)
except KeyboardInterrupt:
    GPIO.cleanup()
```

Make sure that the print statement is indented inside the **if** statement. If not, the value of **button_count** will be printed every time the main loop runs, which is very, very often.

STEP #7

Since this program will be checking **GPIO13** very often, it will consume more resources than it really needs. Adding a delay of **0.05** seconds to the main loop will slow down the button checks a small amount, while drastically reducing the CPU load:

```
try:
    while True:
        if GPIO.input(13) == False:
            button_count = button_count + 1
            print(button_count)
        time.sleep(0.05)
except KeyboardInterrupt:
```

Make sure the indentation on this delay is aligned with the **if:** block above so that it runs even if the **if:** block does not get triggered by **GPIO13**.

STEP #8

Run the program and press the button a few times. Do you notice anything about the count? It's not very accurate, at all! You didn't add any code to slow down the checking of **GPIO13** which is happening about 1000 times per second. Every time that **GPIO13** is checked, and found to be low (button is pressed), **button_count** is incremented by 1 and the value is printed to the console. You will need to tune up this loop with a delay to make it run more accurately, which you will do in the next activity.

Below is a copy of the whole program for your reference:

```
import RPi.GPIO as GPIO, time

GPIO.setmode(GPIO.BCM)

GPIO.setup(13, GPIO.IN, pull_up_down=GPIO.PUD_UP)


button_count = 0

try:

    while True:

        if GPIO.input(13) == False:

            button_count = button_count + 1

            print(button_count)

        time.sleep(0.05)

except KeyboardInterrupt:

    GPIO.cleanup()
```

ACTIVITY #3 – TUNING A LOOP USING A DELAY

In this activity, you will modify the program from the last activity to make it count button presses more accurately.

STEP #1

You will be adding a delay to the main program to slow down how often the state of GPIO13 is checked. Use a delay value of 2 seconds and see how the program responds. Add a **`time.sleep(2)`** just below the print statement:

```
try:
    while True:
        if GPIO.input(13) == False:
            button_count = button_count + 1
            print(button_count)
            time.sleep(2)
         time.sleep(0.05)
except KeyboardInterrupt:
    GPIO.cleanup()
```

STEP #2

Run the program and press the button a few times. Pressing the button results in **button_count** being incremented by 1, the new value of **button_count** being printed to the console, and then a two second delay. This delay works well to slow down the loop and make counts more accurate, but the program feels a little unresponsive during those large two second delays when new presses are ignored.

Reduce the value of the delay from **2** down to **0.2**:

```
try:
    while True:
        if GPIO.input(13) == False:
            button_count = button_count + 1
            print(button_count)
            time.sleep(0.2)
        time.sleep(0.05)
except KeyboardInterrupt:
    GPIO.cleanup()
```

STEP #3

Run the program again with the new delay value and press the button a few times. You will find that a value of 0.2 for the delay is just right for keeping the program responsive after a button press, as well as eliminating a single press being counted multiple times.

The word "tuning" was used in title of this section because a loop might need slight timing changes based on how it's causing your program to behave. No delay caused multiple counts, while 2 full seconds made the program feel unresponsive. For this specific program, a delay of 0.2 was a good balance between those two to ensure the most accurate counting performance from the program.

ACTIVITY #4 – ADDING LED CONFIRMATION USING THE SLIDE SWITCH

In this activity, you will modify the program from the last activity to flash the LED each time the value of **button_count** is incremented. Input from the slide switch will be used to enable or disable the LED indicator.

STEP #1

The GPIO pins for the slide switch and LED will need to be configured so they can be used in this program. **GPIO22** will need to be configured as an active-low input (software pull-up required), and **GPIO26** will need to be an output. Using your program from the last activity as a starting point, make these additions just below the existing **GPIO.setup** line:

```
GPIO.setup(13, GPIO.IN, pull_up_down=GPIO.PUD_UP)
GPIO.setup(22, GPIO.IN, pull_up_down=GPIO.PUD_UP)
GPIO.setup(26, GPIO.OUT)

button_count = 0
```

STEP #2

The main loop will already be running and checking the pushbutton switch on **`GPIO13`**. We only want the LED to come on when an increment event happens, so we can nest the LED code inside of the existing if block. After the value is **`button_count`** is printed, add an **`if`** condition that will check to see if **`GPIO22`** is low or **`False`**, and if so, turn on the LED at **`GPIO26`**:

```
try:
    while True:
        if GPIO.input(13) == False:
            button_count = button_count + 1
            print(button_count)
            if GPIO.input(22) == False:
                GPIO.output(26, GPIO.HIGH)
            time.sleep(0.2)
```

The time delay from the last lesson can now perform the additional task of keeping the LED on for 0.2 seconds, which is a good quick flash for an indicator light.

STEP #3

The only thing missing is a way to turn the LED off. We can fix this by adding a **GPIO.output** line to the end of the main loop that will turn off the LED:

```
try:
    while True:
        if GPIO.input(13) == False:
            button_count = button_count + 1
            print(button_count)
            if GPIO.input(22) == False:
                GPIO.output(26, GPIO.HIGH)
            time.sleep(0.2)
            GPIO.output(26, GPIO.LOW)
```

The LED will be turned off every time the main loop runs. If the slide switch is applying ground to **GPIO22** when the main loop runs, then the LED will be turned on, delayed for 0.2 seconds, and then turned off at the end of the main loop.

If the slide switch is not applying a ground to **GPIO22** when the main loop runs, the LED will not turn on, the program will still delay 0.2 seconds, and then the LED will be told to turn off, even though it's already off. Sending a **GPIO.LOW** command to a GPIO pin that's already low will not harm anything. This just ensures that the LED gets turned off at the end of the main loop, regardless of whether it happens to be on or off.

STEP #4

Run the program. Press the pushbutton to accumulate some counts, and then move the slide switch to the opposite position, and verify the LED reacts as expected. With the slide switch selector closest to the pushbutton switch, counts will be indicated by the LED flash, With the slide switch in the opposite position, the LED will stay off, but counts will continue to accumulate and be printed when the pushbutton is pressed.

A copy of the entire program is available on the next page for reference.

Leave the circuit assembled for use in Lesson B-6.

```
import RPi.GPIO as GPIO, time
GPIO.setmode(GPIO.BCM)
GPIO.setup(13, GPIO.IN, pull_up_down=GPIO.PUD_UP)
GPIO.setup(22, GPIO.IN, pull_up_down=GPIO.PUD_UP)
GPIO.setup(26, GPIO.OUT)

button_count = 0

try:
    while True:
        if GPIO.input(13) == False:
            button_count = button_count + 1
            print(button_count)
            if GPIO.input(22) == False:
                GPIO.output(26, GPIO.HIGH)
            time.sleep(0.2)
            GPIO.output(26, GPIO.LOW)
        time.sleep(0.05)

except KeyboardInterrupt:
    GPIO.cleanup()
```

QUESTIONS FOR UNDERSTANDING

1. Can pull-ups and pull-downs be added within your program, or does every input switch require connections to physically pull-up or pull-down resistors?

2. Why is switch contact bounce a problem for programs that are counting switch presses?

3. Does an active-low input need to be connected to 3.3V or ground to trigger that input?

Answers can be found on the next page.

ANSWERS TO THE QUESTIONS FOR UNDERSTANDING

1. *Can pulling inputs up and down be done within your program, or does every switch require connections to physically pull-up or pull-down resistors?*

   ANSWER: Pulling inputs up and down can be done within your program, when you configure the GPIO pin as an input.

2. *Why is switch contact bounce a problem for programs that are counting switch presses?*

   ANSWER: If your application for counting switch contact is very accurate like a computer and can register multiple switch contacts per second, so a single button press could result in several switch counts, rather than only one being counted.

3. *Does an active-low input need to be connected to 3.3V or ground to trigger that input?*

   ANSWER: An active-low input means that the input must be low to activate the input. That input will need a ground applied to over come the pull-up and activate the input in software.

CONCLUSION

In this lesson you learned to work with both slide and pushbutton switches as well as how to account for switch bounce.

In the next lesson, you will learn to work with logical operators that will allow you to create programs with more advanced behavior based on multiple GPIO inputs or variables.