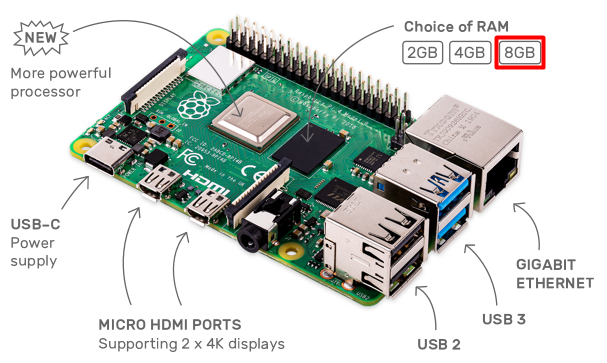
# Objective

Write Python code to read the status of KEY and further to control LED.

1. Learn to read the status of Raspi IO input port；
2. Learn the Python function and interrupt programing；
3. Detect click and hold to control LED.

# Components

1. Raspi Board and Power；



1. A 16GB SD Card and a SD Card Reader；

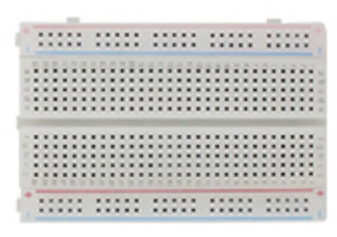


1. HDMI Screen, USB keyboard, USB mouse and HDMI Cable

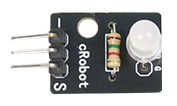




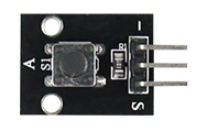
1. Basic Hardware (Adapter Board, Wires, Bread Board)



1. Double color LED



1. KEY



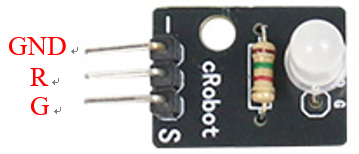
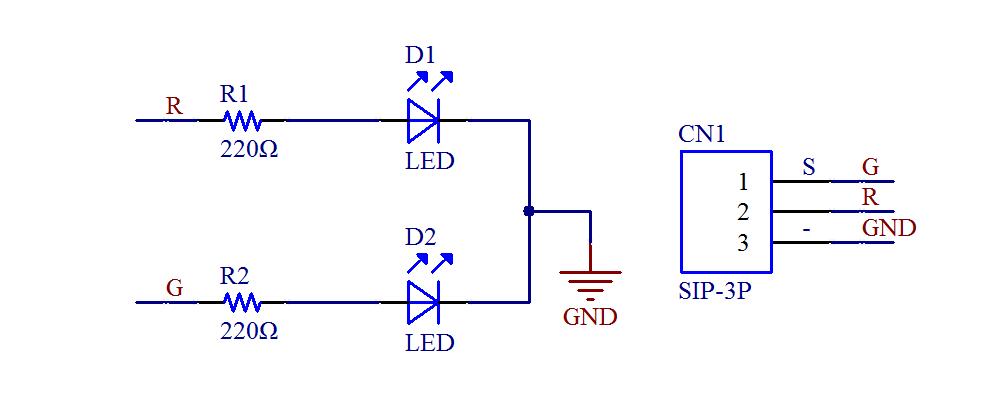
1. Capacitive TOUCH switch



# Basic Knowledge for the Experiment

## LED

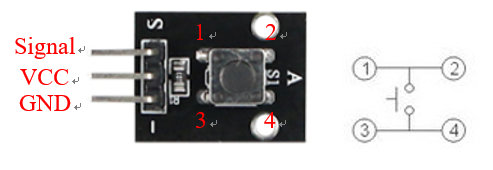
The hardware device and the corresponding schematic of double color LED is shown as below:

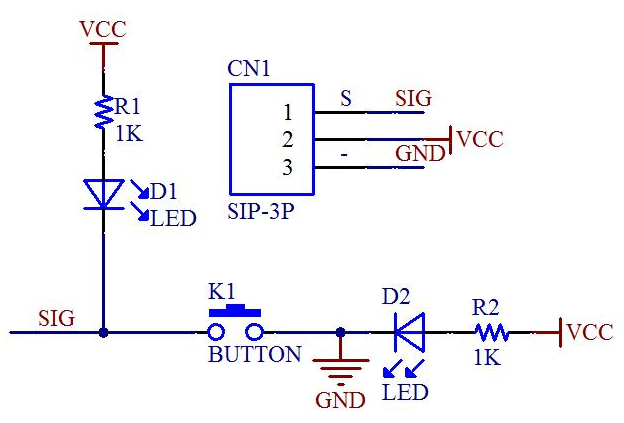
 

There are two LED luminous source, which emit red and green light, respectively. If Raspi outputs a high level to **‘R’** pin, the red LED on; If Raspi outputs a low level to **‘R’** pin, while a high level to **‘G’** pin, the green LED on. If both **‘R’** and **‘G’** pins are high level, only the red LED on.

## KEY

The hardware device and the corresponding schematic of KEY is shown as below:

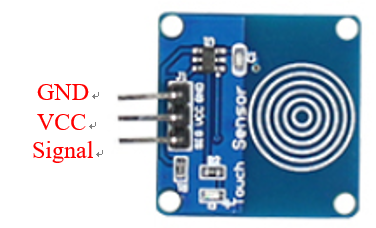


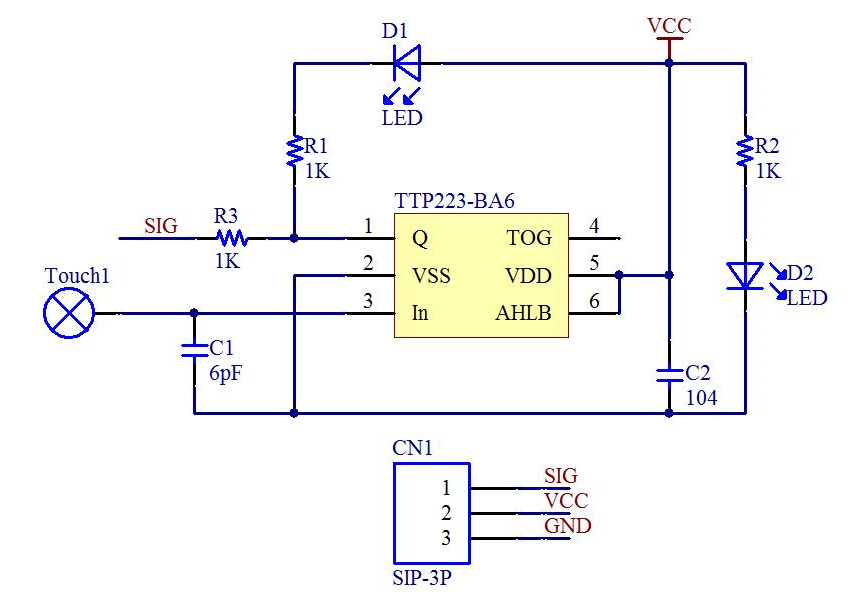


When the button is pressed, the **‘Signal (SIG)’** pin outputs **low** voltage (0V), else the **‘Signal (SIG)’** pin outputs **high** voltage (3.3V).

## TOUCH switch

The capacitive touch switch is similar with KEY, while the KEY is replaced by a induction zone.





Compared with KEY, when we touch the induction zone of TOUCH switch, the **‘Signal (SIG)’** pin outputs **high** voltage (3.3V), else the **‘Signal (SIG)’** pin outputs **low** voltage (0V).

## How to Obtain the Status of Key

There are two main ways to check input pin states: polling and interrupts.

Here’s the difference through a real life example: imagine you’re waiting for an important email and want to open it as soon as it arrives. You can either:

* Check your emails every 5 minutes, or 1 minutes, or even 10 seconds until you get it.
* Or you can activate a notification bell so you get a popup on your screen as soon as the email arrives.

As you can guess, the second method is much more efficient in this case. When dealing with interrupts, think “notification”. Note that interrupts (notifications) are not the solution to all problems, it really depends on the situation.

How Raspberry Pi GPIO interrupts work ?

Interrupts will be triggered when a signal’s state (LOW/HIGH) changes.



There are 2 kind of interrupts:

FALLING: when the state goes from HIGH to LOW, i.e. KEY is pressed.

RISING: when the state goes from LOW to HIGH, i.e. KEY is released.

So, basically in your program if you set up a FALLING (pressed) interrupt, then your program will be executed as soon as the KEY is pressed. Otherwise, if you set up a RISING (released) interrupt, then your program will be executed as soon as the KEY is released.

KEY jitter：Identically, during the process of pressing key to releasing key, both the falling edge and rising edge will only be generated once. While in the actual situation, the level change is different from the theoretical change. Because the KEYs are mechanical elastic switches. When the mechanical contact is opened and closed, due to the elastic action of the mechanical contact, a key switch will not be steadily turned on immediately when it is closed, and also will not be completely opened immediately when it is opened. It was accompanied by a series of jitters in an instant. The difference is show as below:



In order to ensure that the program only responds once to a key closure or opening, the key must be debounced, that is, when the key state change is detected, the program will not respond immediately, but wait for the closure or opening to stabilize before processing. The jitter time is determined by the mechanical characteristics of the KEYs, generally 10~100ms.

Actually, the button class of **gpiozero** library has the function of removing jitter, if it not work well, you can remove it by yourself.

## Python Function and class

### **LED** class of **gpiozero** library

**class** LED**(pin, \*, active\_high=True, initial\_value=False, pin\_factory=None)** [[source code]](https://gpiozero.readthedocs.io/en/stable/_modules/gpiozero/output_devices.html#LED)

|  |  |
| --- | --- |
| **Parameters:** | * **pin** ([*int*](https://docs.python.org/3.7/library/functions.html#int) *or* [*str*](https://docs.python.org/3.7/library/stdtypes.html#str)) – The GPIO pin which the LED is connected to. See [Pin Numbering](https://gpiozero.readthedocs.io/en/stable/recipes.html#pin-numbering) for valid pin numbers. If this is [**None**](https://docs.python.org/3.7/library/constants.html#None) a [**GPIODeviceError**](https://gpiozero.readthedocs.io/en/stable/api_exc.html#gpiozero.GPIODeviceError) will be raised. * **active\_high** ([*bool*](https://docs.python.org/3.7/library/functions.html#bool)) – If [**True**](https://docs.python.org/3.7/library/constants.html#True) (the default), the LED will operate normally with the circuit described above. If [**False**](https://docs.python.org/3.7/library/constants.html#False) you should wire the cathode to the GPIO pin, and the anode to a 3V3 pin (via a limiting resistor). * **initial\_value** ([*bool*](https://docs.python.org/3.7/library/functions.html#bool) *or* [*None*](https://docs.python.org/3.7/library/constants.html#None)) – If [**False**](https://docs.python.org/3.7/library/constants.html#False) (the default), the LED will be off initially. If [**None**](https://docs.python.org/3.7/library/constants.html#None), the LED will be left in whatever state the pin is found in when configured for output (warning: this can be on). If [**True**](https://docs.python.org/3.7/library/constants.html#True), the LED will be switched on initially. * **pin\_factory** ([*Factory*](https://gpiozero.readthedocs.io/en/stable/api_pins.html#gpiozero.Factory) *or* [*None*](https://docs.python.org/3.7/library/constants.html#None)) – See [API - Pins](https://gpiozero.readthedocs.io/en/stable/api_pins.html) for more information (this is an advanced feature which most users can ignore). |

off()

Turns the device off.

on()

Turns the device on.

toggle()

Reverse the state of the device. If it’s on, turn it off; if it’s off, turn it on.

is\_lit

Returns [**True**](https://docs.python.org/3.7/library/constants.html#True) if the device is currently active and [**False**](https://docs.python.org/3.7/library/constants.html#False) otherwise. This property is usually derived from [**value**](https://gpiozero.readthedocs.io/en/stable/api_output.html#gpiozero.LED.value). Unlike [**value**](https://gpiozero.readthedocs.io/en/stable/api_output.html#gpiozero.LED.value), this is always a boolean.

pin

The [**Pin**](https://gpiozero.readthedocs.io/en/stable/api_pins.html#gpiozero.Pin) that the device is connected to. This will be [**None**](https://docs.python.org/3.7/library/constants.html#None) if the device has been closed (see the [**close()**](https://gpiozero.readthedocs.io/en/stable/api_generic.html#gpiozero.Device.close) method). When dealing with GPIO pins, query pin.number to discover the GPIO pin (in BCM numbering) that the device is connected to.

value

Returns 1 if the device is currently active and 0 otherwise. Setting this property changes the state of the device.

For more information about output devices, please click [here](https://gpiozero.readthedocs.io/en/stable/api_output.html#led).

### **Button** class of **gpiozero** library

***class*** gpiozero.Button**(*pin*,*\**, *pull\_up=True*, *active\_state=None*, *bounce\_time=None*, *hold\_time=1*, *hold\_repeat=False*, *pin\_factory=None*)** [[source code]](https://gpiozero.readthedocs.io/en/stable/_modules/gpiozero/input_devices.html#Button)

Extends [DigitalInputDevice](https://gpiozero.readthedocs.io/en/stable/api_input.html#gpiozero.DigitalInputDevice) and represents a simple push button or switch.

Connect one side of the button to a ground pin, and the other to any GPIO pin. Alternatively, connect one side of the button to the 3V3 pin, and the other to any GPIO pin, then set *pull\_up* to [**False**](https://docs.python.org/3.7/library/constants.html#None) in the [Button](https://gpiozero.readthedocs.io/en/stable/api_input.html#gpiozero.Button) constructor.

|  |  |
| --- | --- |
| **Parameters:** | * **pin** ([*int*](https://docs.python.org/3.7/library/functions.html#int) *or* [*str*](https://docs.python.org/3.7/library/stdtypes.html#str)) – The GPIO pin which the button is connected to. See [Pin Numbering](https://gpiozero.readthedocs.io/en/stable/recipes.html#pin-numbering) for valid pin numbers. If this is [**None**](https://docs.python.org/3.7/library/constants.html#None) a [**GPIODeviceError**](https://gpiozero.readthedocs.io/en/stable/api_exc.html#gpiozero.GPIODeviceError) will be raised. * **pull\_up** ([*bool*](https://docs.python.org/3.7/library/functions.html#bool) *or* [*None*](https://docs.python.org/3.7/library/constants.html#None)) – If [**True**](https://docs.python.org/3.7/library/constants.html#True) (the default), the GPIO pin will be pulled high by default. In this case, connect the other side of the button to ground. If [**False**](https://docs.python.org/3.7/library/constants.html#False), the GPIO pin will be pulled low by default. In this case, connect the other side of the button to 3V3. If [**None**](https://docs.python.org/3.7/library/constants.html#None), the pin will be floating, so it must be externally pulled up or down and the active\_state parameter must be set accordingly. * **active\_state** ([*bool*](https://docs.python.org/3.7/library/functions.html#bool) *or* [*None*](https://docs.python.org/3.7/library/constants.html#None)) – See description under [**InputDevice**](https://gpiozero.readthedocs.io/en/stable/api_input.html#gpiozero.InputDevice) for more information. * **bounce\_time** ([*float*](https://docs.python.org/3.7/library/functions.html#float) *or* [*None*](https://docs.python.org/3.7/library/constants.html#None)) – If [**None**](https://docs.python.org/3.7/library/constants.html#None) (the default), no software bounce compensation will be performed. Otherwise, this is the length of time (in seconds) that the component will ignore changes in state after an initial change. * **hold\_time** ([*float*](https://docs.python.org/3.7/library/functions.html#float)) – The length of time (in seconds) to wait after the button is pushed, until executing the [**when\_held**](https://gpiozero.readthedocs.io/en/stable/api_input.html#gpiozero.Button.when_held) handler. Defaults to 1. * **hold\_repeat** ([*bool*](https://docs.python.org/3.7/library/functions.html#bool)) – If [**True**](https://docs.python.org/3.7/library/constants.html#True), the [**when\_held**](https://gpiozero.readthedocs.io/en/stable/api_input.html#gpiozero.Button.when_held) handler will be repeatedly executed as long as the device remains active, every *hold\_time* seconds. If [**False**](https://docs.python.org/3.7/library/constants.html#False) (the default) the [**when\_held**](https://gpiozero.readthedocs.io/en/stable/api_input.html#gpiozero.Button.when_held) handler will be only be executed once per hold. * **pin\_factory** ([*Factory*](https://gpiozero.readthedocs.io/en/stable/api_pins.html#gpiozero.Factory) *or* [*None*](https://docs.python.org/3.7/library/constants.html#None)) – See [API - Pins](https://gpiozero.readthedocs.io/en/stable/api_pins.html) for more information (this is an advanced feature which most users can ignore). |

wait\_for\_press(timeout=None)

Pause the script until the device is activated, or the timeout is reached. **timeout** ([float](https://docs.python.org/3.7/library/functions.html#float) or [None](https://docs.python.org/3.7/library/constants.html#None)) – Number of seconds to wait before proceeding. If this is [**None**](https://docs.python.org/3.7/library/constants.html#None) (the default), then wait indefinitely until the device is active.

wait\_for\_release(timeout=None)

Pause the script until the device is deactivated, or the timeout is reached. **timeout** ([float](https://docs.python.org/3.7/library/functions.html#float) or [None](https://docs.python.org/3.7/library/constants.html#None)) – Number of seconds to wait before proceeding. If this is [**None**](https://docs.python.org/3.7/library/constants.html#None) (the default), then wait indefinitely until the device is inactive.

held\_time

The length of time (in seconds) that the device has been held for. This is counted from the first execution of the [**when\_held**](https://gpiozero.readthedocs.io/en/stable/api_input.html#gpiozero.Button.when_held) event rather than when the device activated, in contrast to [**active\_time**](https://gpiozero.readthedocs.io/en/stable/api_generic.html#gpiozero.EventsMixin.active_time). If the device is not currently held, this is [**None**](https://docs.python.org/3.7/library/constants.html#None).

hold\_repeat

If [**True**](https://docs.python.org/3.7/library/constants.html#True), [**when\_held**](https://gpiozero.readthedocs.io/en/stable/api_input.html#gpiozero.Button.when_held) will be executed repeatedly with [**hold\_time**](https://gpiozero.readthedocs.io/en/stable/api_input.html#gpiozero.Button.hold_time) seconds between each invocation.

hold\_time

The length of time (in seconds) to wait after the device is activated, until executing the [**when\_held**](https://gpiozero.readthedocs.io/en/stable/api_input.html#gpiozero.Button.when_held) handler. If [**hold\_repeat**](https://gpiozero.readthedocs.io/en/stable/api_input.html#gpiozero.Button.hold_repeat) is True, this is also the length of time between invocations of [**when\_held**](https://gpiozero.readthedocs.io/en/stable/api_input.html#gpiozero.Button.when_held).

is\_held

When [**True**](https://docs.python.org/3.7/library/constants.html#True), the device has been active for at least [**hold\_time**](https://gpiozero.readthedocs.io/en/stable/api_input.html#gpiozero.Button.hold_time) seconds.

is\_pressed

Returns [**True**](https://docs.python.org/3.7/library/constants.html#True) if the device is currently active and [**False**](https://docs.python.org/3.7/library/constants.html#False) otherwise. This property is usually derived from [**value**](https://gpiozero.readthedocs.io/en/stable/api_input.html#gpiozero.Button.value). Unlike [**value**](https://gpiozero.readthedocs.io/en/stable/api_input.html#gpiozero.Button.value), this is always a boolean.

pin

The [**Pin**](https://gpiozero.readthedocs.io/en/stable/api_pins.html#gpiozero.Pin) that the device is connected to. This will be [**None**](https://docs.python.org/3.7/library/constants.html#None) if the device has been closed (see the [**close()**](https://gpiozero.readthedocs.io/en/stable/api_generic.html#gpiozero.Device.close) method). When dealing with GPIO pins, query pin.number to discover the GPIO pin (in BCM numbering) that the device is connected to.

pull\_up

If [**True**](https://docs.python.org/3.7/library/constants.html#True), the device uses a pull-up resistor to set the GPIO pin “high” by default.

value

Returns 1 if the button is currently pressed, and 0 if it is not.

when\_held

The function to run when the device has remained active for [**hold\_time**](https://gpiozero.readthedocs.io/en/stable/api_input.html#gpiozero.Button.hold_time) seconds. This can be set to a function which accepts no (mandatory) parameters, or a Python function which accepts a single mandatory parameter (with as many optional parameters as you like). If the function accepts a single mandatory parameter, the device that activated will be passed as that parameter. Set this property to [**None**](https://docs.python.org/3.7/library/constants.html#None) (the default) to disable the event.

when\_pressed

The function to run when the device changes state from inactive to active. This can be set to a function which accepts no (mandatory) parameters, or a Python function which accepts a single mandatory parameter (with as many optional parameters as you like). If the function accepts a single mandatory parameter, the device that activated it will be passed as that parameter. Set this property to [**None**](https://docs.python.org/3.7/library/constants.html#None) (the default) to disable the event.

when\_released

The function to run when the device changes state from active to inactive. This can be set to a function which accepts no (mandatory) parameters, or a Python function which accepts a single mandatory parameter (with as many optional parameters as you like). If the function accepts a single mandatory parameter, the device that deactivated it will be passed as that parameter. Set this property to [**None**](https://docs.python.org/3.7/library/constants.html#None) (the default) to disable the event.

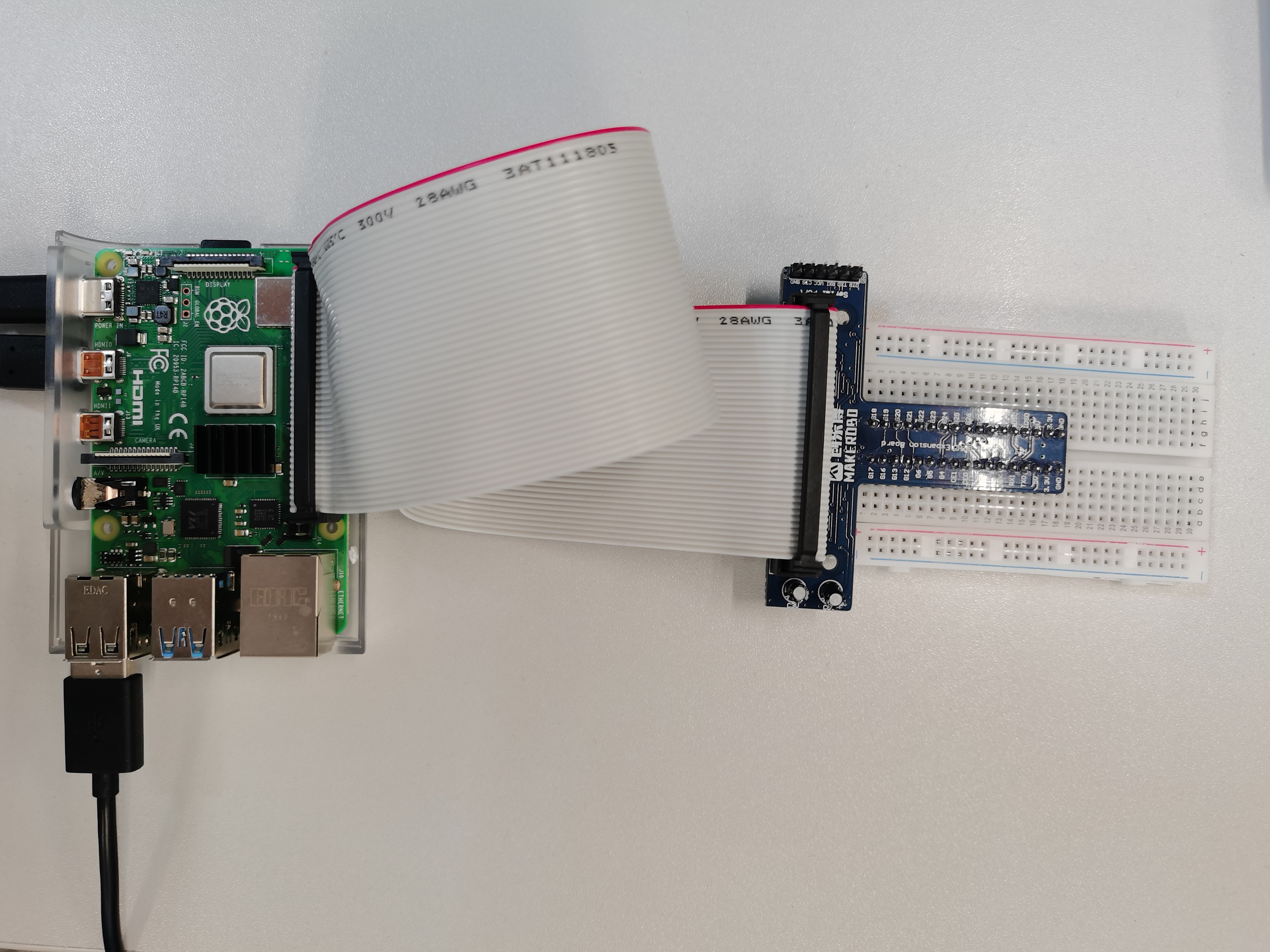
For more information about input devices, please click [here](https://gpiozero.readthedocs.io/en/stable/api_input.html#button).

# Experiment Steps

## Hardware Connection

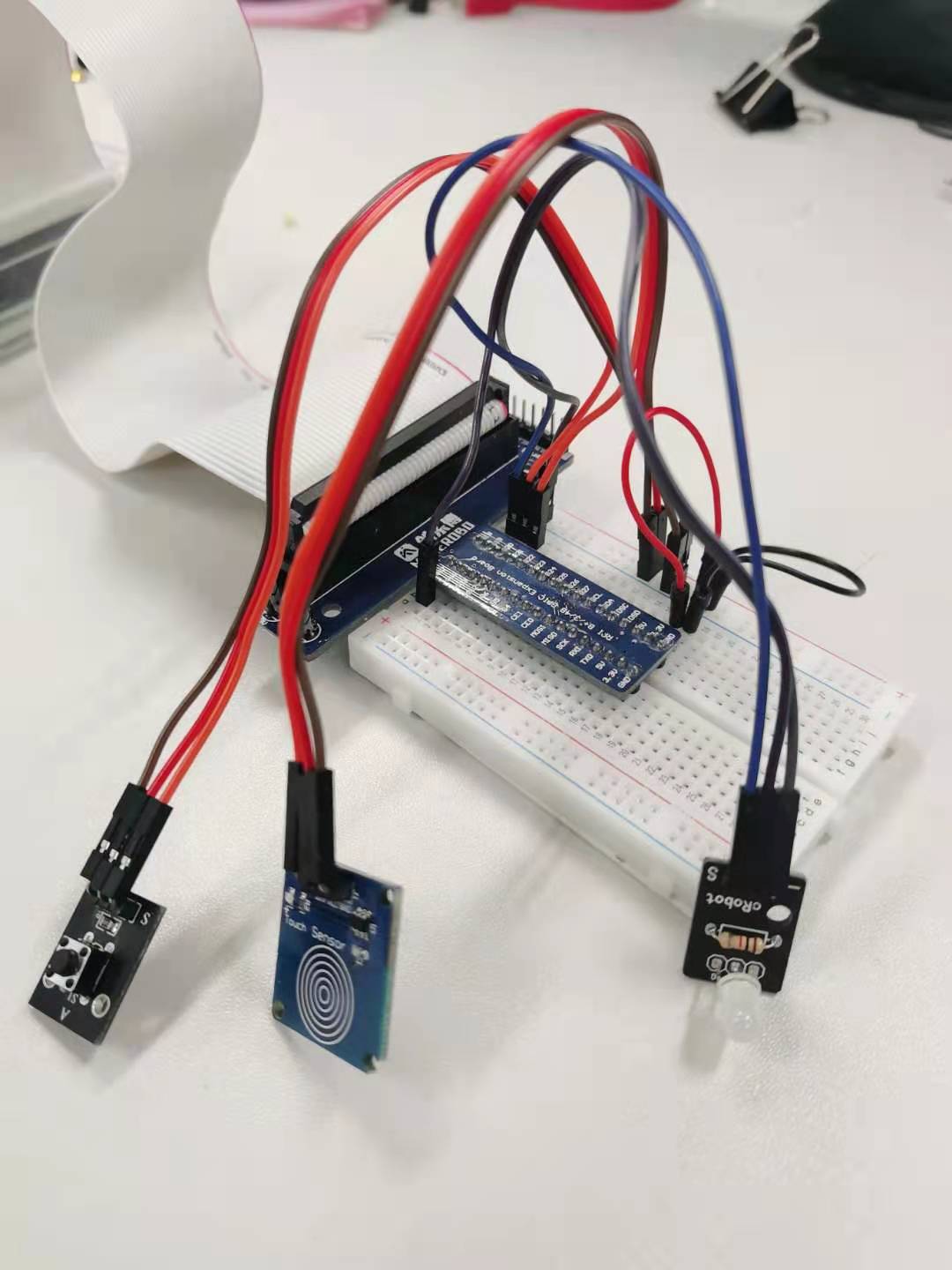
1. Connect Raspi and Breadboard using the 40-wires cable as below.

This side close to the edge of Raspi



1. Connect LED, KEY and Capacitive TOUCH switch as below.

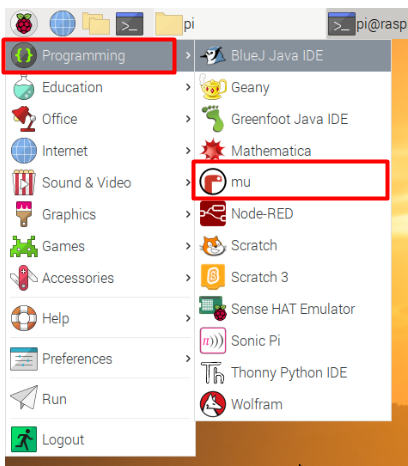




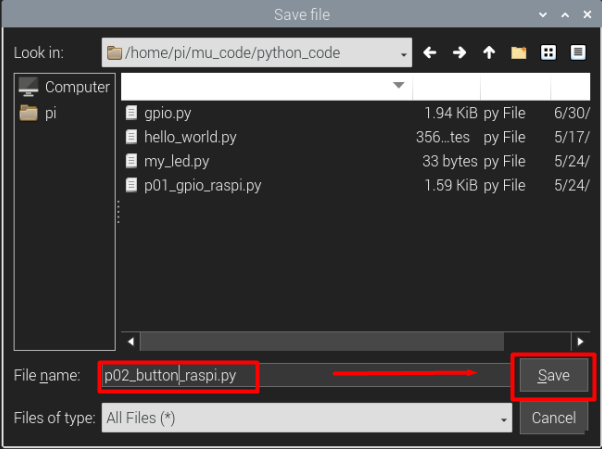
## Software Programing

### Create New File

1. Power on Raspi and start-up Mu from the Raspi menu: **Menu 🡪 Programming 🡪 mu**, as below



1. Click the New button, then before typing anything, click the Save button and enter the “python\_code” folder, and give your file the name, “p02\_button\_raspi.py”, then click Save button.



### Initial Program

1. Type the following Python code into the text area to import the useful python libraries:

from gpiozero import \*

from time import sleep

from time import time

1. Use class Button() to initialize KEY and TOUCH switch.

# Set GPIO19 as KEY input, Low voltage is active state

pin\_key = Button(19, pull\_up=None, active\_state=False)

# Set GPIO20 as TOUCH switch input, High voltage is active state

pin\_tou = Button(20, pull\_up=None, active\_state=True)

As the **‘Signal (SIG)’** pin of KEY board outputs **low** voltage (0V) when we press, so we set active\_state=False, while active\_state=True for TOUCH switch due to the **‘Signal (SIG)’** pin of TOUCH board outputs **high** voltage (3.3V) when we press

1. Use class LED() to initialize the red LED and green LED.

led\_red = LED(17) # Set GPIO17 as red LED

led\_gre = LED(18) # Set GPIO18 as green LED

### Polling Method

1. Write the function key\_poll\_test() as bellow to poll the status of KEY and TOUCH switch.

def key\_poll\_test():

print('Polling Mode !')

while True:

if pin\_key.is\_pressed is True: # KEY is pressed

led\_red.on()

sleep(0.1)

led\_red.off()

sleep(0.1)

if pin\_tou.is\_pressed is True: # TOUCH switch is pressed

led\_gre.on()

sleep(0.1)

led\_gre.off()

sleep(0.1)

if pin\_key.is\_pressed & pin\_tou.is\_pressed:

print('Polling test end !!!')

return True

A **function** is a block of code which only runs when it is called. You can pass data, known as parameters, into a **function**. A **function** can return data as a result.

In this function, we poll the status of KEY and TOUCH switch. Each button are pressed the corresponding LED will flash. if pin\_key.is\_pressed & pin\_tou.is\_pressed: means that if both pin\_key.is\_pressed and pin\_tou.is\_pressed are true, the following program will be executed. The last sentence return True means the function is finished and the program will jump out the current function.

1. At the end of p02\_button\_raspi.py, write the following code to run key\_poll\_test().

# These program only executed in this manuscript

if \_\_name\_\_ == '\_\_main\_\_':

while True:

print('Please input mode:')

in\_data = input() # input data from keyboard

if in\_data == 'P': # when 'P' is inputted

key\_poll\_test()

elif in\_data == 'exit':

break

if \_\_name\_\_ == '\_\_main\_\_': means the program only executed in this manuscript

while True: means the program always be executed repeatedly (endless loop).

in\_data = input() obtains input chars from Keyboard and store chars into in\_data. If we input P, key\_poll\_test() will be executed. While if we input exit, the program will jump out the endless loop.

1. Press the Run button. At the output area, a message, Please input mode:, will be output as shown below. Then we input P in the output area and press Enter, the key\_poll\_test() will be executed and a message, Polling Mode !, will be output that means the function key\_poll\_test() is running now.



1. Now if we press KEY, the red LED should flash. If we press TOUCH switch, the green LED should flash.
2. Press both the KEY and TOUCH switch, the key\_poll\_test() will output Polling test end !!! and exit. Once key\_poll\_test() is exited, the KEY and TOUCH switch can control LED any more. And the output area will remain us select mode again.



### Interrupt Method

1. Write the interrupt response functions for KEY and TOUCH switch.

def key\_pressed\_led(): # interrupt function when press KEY

print('Key is pressed')

led\_red.on()

sleep(0.1)

led\_red.off()

def tou\_pressed\_led(): # interrupt function when press TOUCH switch

print('Touch switch is pressed')

led\_gre.on()

sleep(0.1)

led\_gre.off()

1. Write the function key\_intr\_test() as below to associate the KEY and TOUCH switch with key\_pressed\_led() and tou\_pressed\_led(), respectively. Further, if both KEY and TOUCH switch are pressed, the association will be removed and the program will jump out the current function.

def key\_intr\_test():

print('Interrupt Mode !')

# register interrupt function for KEY pressing

pin\_key.when\_pressed = key\_pressed\_led

# register interrupt function for TOUCH switch pressing

pin\_tou.when\_pressed = tou\_pressed\_led

while True: # poll status of two buttons

sleep(0.1)

if pin\_key.is\_pressed & pin\_tou.is\_pressed:

led\_gre.blink(n=1) # LED blink to show program end

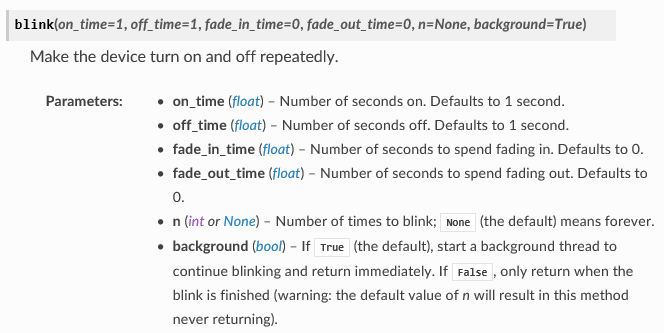
print('Interrupt test end !!!')

pin\_key.when\_pressed = None # Remove interrupt function

pin\_tou.when\_pressed = None # Remove interrupt function

return True

The highlighted led\_gre.blink(n=1) means green LED flash once. Below is the description about blink() function:



1. At the end of p02\_button\_raspi.py, add two lines of code as below:

if \_\_name\_\_ == '\_\_main\_\_': # These program only executed in this manuscript

while True:

print('Please input mode:')

in\_data = input() # input data from keyboard

if in\_data == 'P': # when 'P' is inputted

key\_poll\_test()

elif in\_data == 'I':

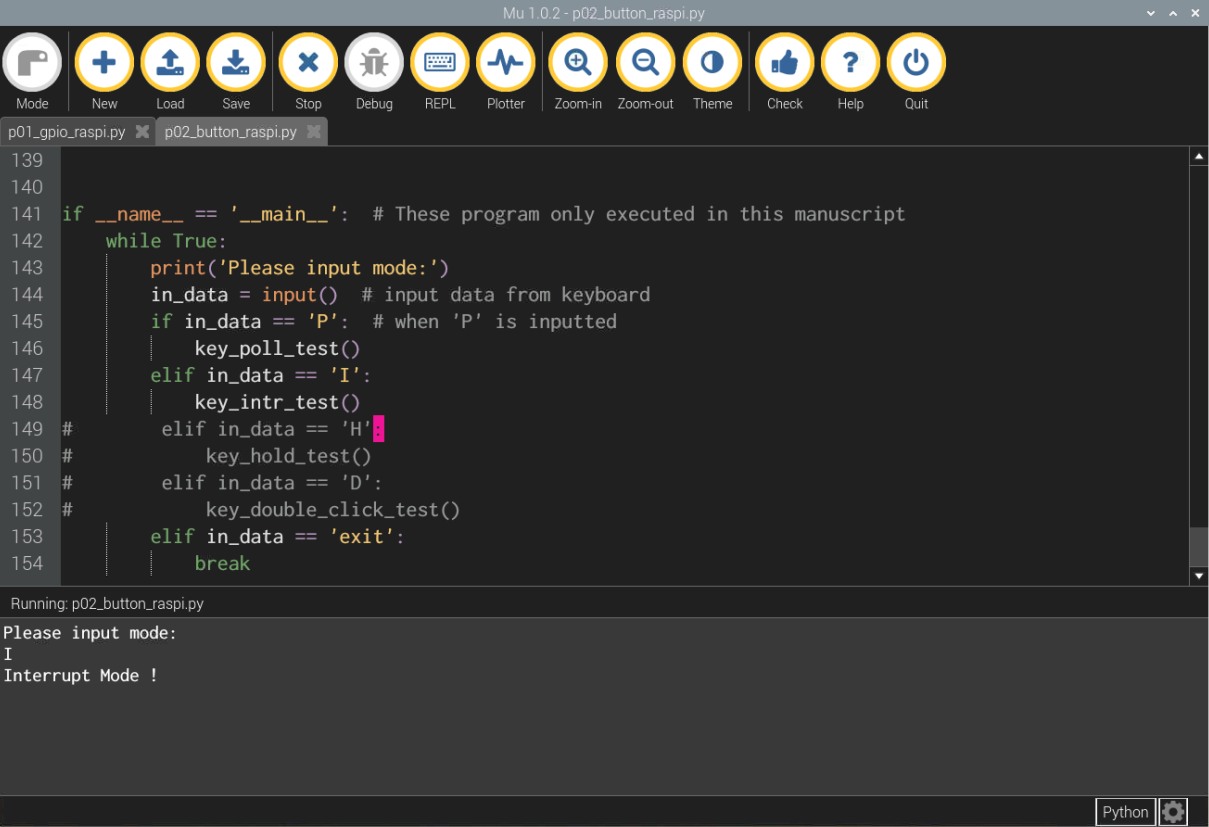
key\_intr\_test()

elif in\_data == 'exit':

break

As highlighted using red coloured words, if we input I, the key\_intr\_test() function will be executed.

1. Press the Run button. Then we input I in the output area and press Enter, the key\_intr\_test() will be executed and a message, Interrupt Mode !, will be output that means the function key\_intr\_test() is running now.



1. Now if we press KEY, the red LED should flash once and the output area print Key is pressed. If we press TOUCH switch, the green LED should flash once and the output area prints Touch switch is pressed.

If the interrupt service function is called more than once when we press KEY each time, it's caused by jitter of KEY. We can increase the sleep time of **key\_pressed\_led()** to resolve this problem.

1. Press both the KEY and TOUCH switch, the key\_intr\_test() will output Interrupt test end !!! and the green LED flash once. Then the program jump out of key\_intr\_test().

### Button Hold

1. Write the interrupt response functions for holding KEY.

def key\_hold\_led():

print('KEY is held')

for i in range(3):

sleep(0.1)

led\_red.on()

sleep(0.1)

led\_red.off()

In this function, the red LED will flash three times. And we can modify the value in for i in range(3): to adjust the repeat time.

1. Write the function key\_hold\_test() as below to associate the KEY with key\_hold\_led() first. Then, we set hold\_time to 1 second, which means the interrupt function will be called when the KEY is pressed and held 1 second. Further, if both KEY and TOUCH switch are held more than 2 seconds, the association will be removed and the program will jump out the current function.

def key\_hold\_test():

print('Hold test!!!')

# register interrupt function for KEY holding

pin\_key.when\_held = key\_hold\_led

pin\_key.hold\_time = 1 # set hold seconds

time\_buf = 0

while True:

sleep(0.1)

if pin\_key.is\_pressed & pin\_tou.is\_pressed:

if time\_buf == 0:

time\_buf = time()

elif time() > time\_buf + 2:

time\_buf = 0

led\_gre.blink(n=1)

print('Hold test end !!!')

pin\_key.when\_held = None

pin\_key.hold\_repeat = False

return True

else:

time\_buf = 0

In this function, time() imported from time library is used to obtain current time. If pin\_key.is\_pressed & pin\_tou.is\_pressed is rising edge, i.e. time\_buf == 0, the program will record current time into time\_buf. When both KEY and TOUCH switch are held more than 2 seconds,i.e. time() > time\_buf + 2, the hold test will be ended and the program will jump out of the current function. Once any button is released in 2 seconds, time\_buf will be set to 0.

1. At the end of p02\_button\_raspi.py, add two lines of code as below:

if \_\_name\_\_ == '\_\_main\_\_': # These program only executed in this manuscript

while True:

print('Please input mode:')

in\_data = input() # input data from keyboard

if in\_data == 'P': # when 'P' is inputted

key\_poll\_test()

elif in\_data == 'I':

key\_intr\_test()

elif in\_data == 'H':

key\_hold\_test()

elif in\_data == 'exit':

break

As highlighted using red coloured words, if we input H, the key\_hold\_test() function will be executed.

1. Press the Run button. Then we input H in the output area and press Enter, the key\_hold\_test() will be executed and a message, Hold test !!!, will be output that means the function key\_hold\_test() is running now.



1. Now if we press KEY and hold 1 seconds, the red LED should flash three times and the output area print KEY is held.
2. Press both the KEY and TOUCH switch and hold more than 2 seconds, the key\_hold\_test() will output Hold test end !!! and the green LED flash once. Then the program jump out of key\_hold\_test().

# Task

There are a total of three tasks, you should finish at least two tasks.

1. Modify key\_poll\_test() function, and make the LED only flash once no matter how long the KEY are pressed and held.

You should define a new variable to record the status of KEY.

1. Refer to 4.2.5. , write a new interrupt service function, named tou\_hold\_led(). Modify key\_hold\_test() function to associate TOUCH switch with tou\_hold\_led(). If the TOUCH switch are pressed and held 2 seconds, the tou\_hold\_led() are called.

The tou\_hold\_led() has functions as below:

* Print message TOUCH switch hold test !!!;
* Make red and green LED alternatively flash 5 times.

1. Add a double click mode and write the corresponding function, key\_double\_click\_test(), to detect double click of KEY. When the program is started, if we input D in the output area, key\_double\_click\_test() should be executed.

The key\_double\_click\_test() has functions as below:

* If KEY is pressed twice in 1 seconds, red and green LED alternatively flash 2 times;
* If both KEY and TOUCH switch are pressed or the KEY are triple clicked, the test is ended by return True.

The jitter should be removed using sleep(0.1).

# Reference

[**Raspberry Pi GPIO Interrupts Tutorial**](https://roboticsbackend.com/raspberry-pi-gpio-interrupts-tutorial/)

**End of Tutorial 3**