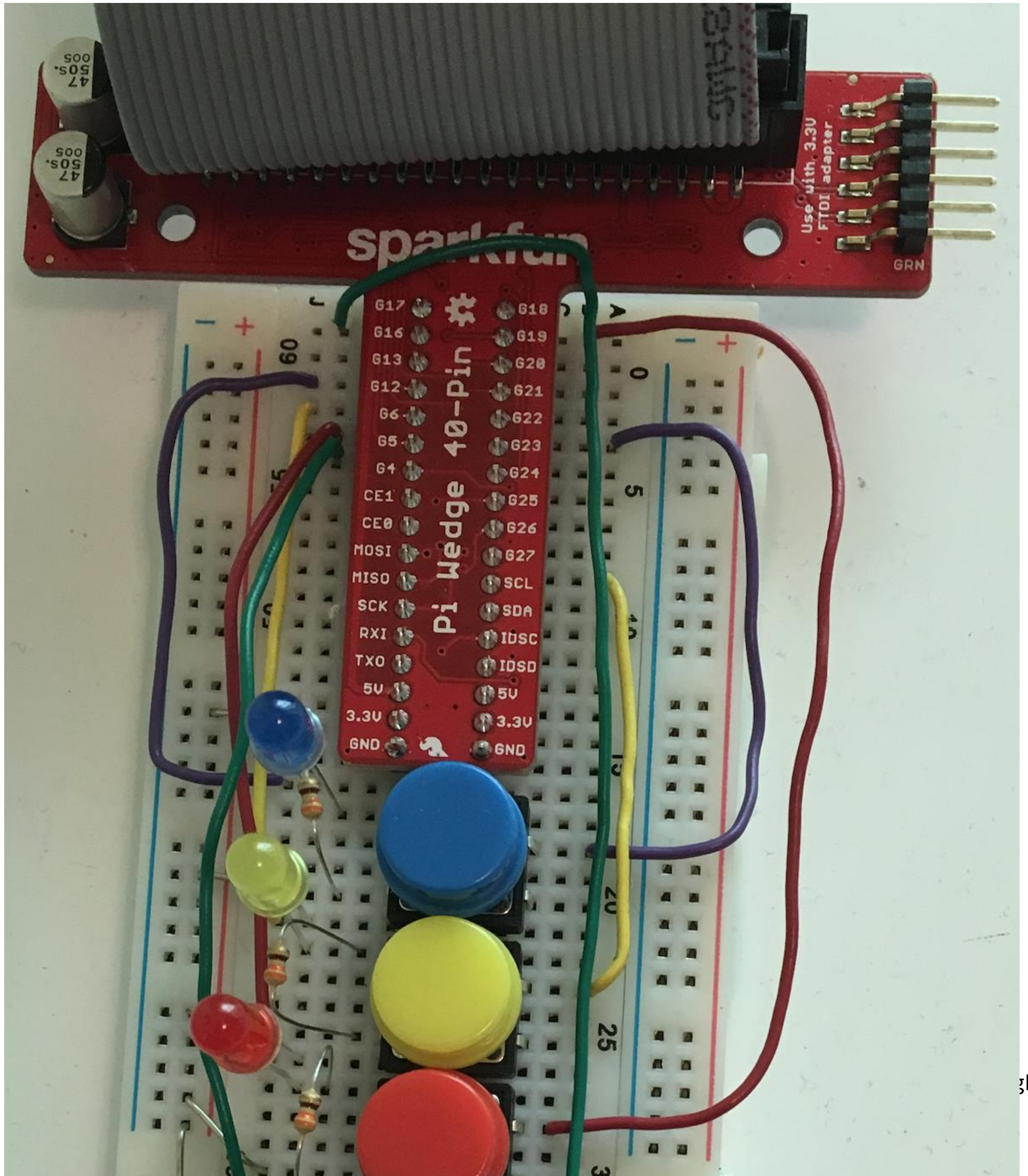
Interrupt-driven I/O on Raspberry Pi 3 with LEDs and pushbuttons: rising/falling edge-detection using RPi.GPIO

*R. X. Seger**Aug 21, 2016*


sparkfun
Pi Wedge 40-pin
Use with 3.3V FTDI adapter
GRN
G17
G16
G13
G12
G6
G5
G4
CE1
CE0
MOSI
MISO
SCK
RXI
TXO
5V
3.3V
GND
G18
G19
G20
G21
G22
G23
G24
G25
G26
G27
SCL
SDA
IDSC
IDSD
5V
3.3V
GND

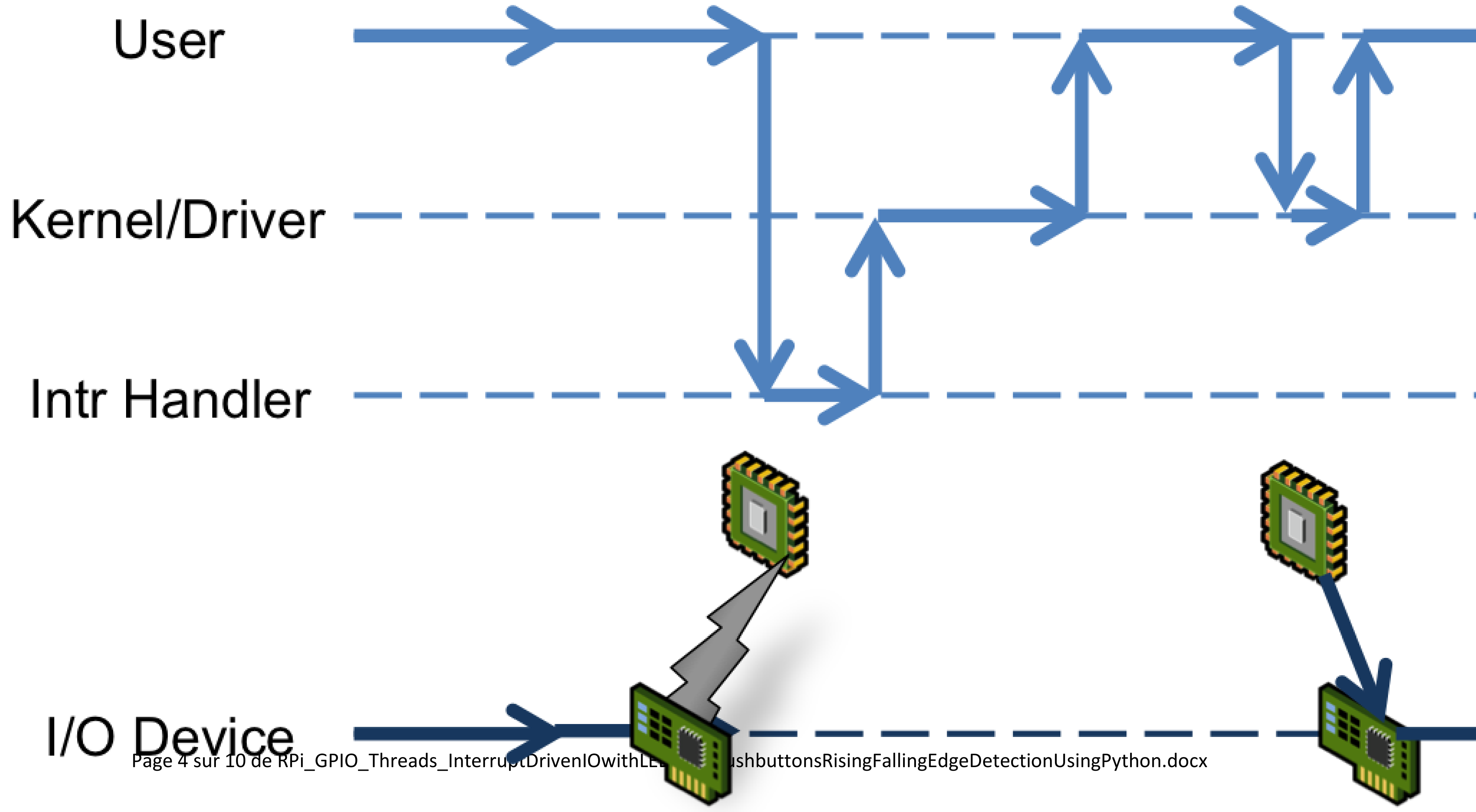
A brief follow-up to *Raspberry Pi 3 GPIO: pushbuttons, LEDs for RC and BARR*, using interrupts instead of polling to read the pushbutton input pins.

First I rewired and repositioned the components on the breadboard as pictured on the left—not needed for changing to interrupts, but to make room for future expansion possibly later. I find it also appears much tidier.

Now for reading from the GPIO port. Starting with gptest.py, the simple test script developed previously to light up LEDs when the corresponding button is pressed. gptest.py used **polling**: calling GPIO.input() in a tight loop, with only a 0.01 second delay between iterations—not very efficient, burning CPU power to continuously loop over and over even though the buttons are rarely if ever pressed.

Consequentially, I only used gptest.py for short periods of time, manually executing it to test the wiring, then terminating soon thereafter. How can this be improved so that the script can run all the time as a daemon?

The solution, invented in 1957 by the great E. W. Dijkstra: **interrupts**.


User
Kernel/Driver
Intr Handler
I/O Device

Interrupt timeline, diagram from virtualirfan's excellent History of Interrupts

Sounds complicated, fortunately the RPi.GPIO Python module included in Raspbian supports interrupts nearly as easily as polling.

Adding callbacks

Raspberrywebserver.com's *Using Interrupt Driven GPIO* is a good introduction. The basic idea is instead of reading the current input state using GPIO.input(), add a callback function using GPIO.add_event_detect().

You can call GPIO.add_event_detect() to say what condition you are looking for, then GPIO.add_event_callback() to set the callback for said condition:

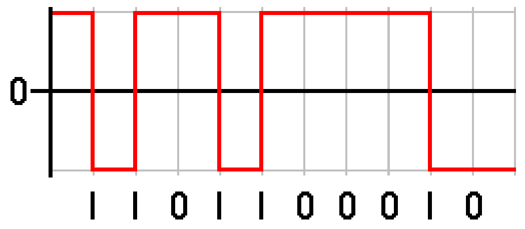
GPIO.add_event_detect(BTN_B, GPIO.RISING)
GPIO.add_event_callback(BTN_B, lambda pin: GPIO.output(LED_B, False))

but it is more straightforward to do it all in one shot using the 3rd argument to GPIO.add_event_detect(), for example:

GPIO.add_event_detect(BTN_B, GPIO.RISING, lambda pin: GPIO.output(LED_B, False))

Edge detection

Both these code snippets do the same thing: changing the state of the blue LED output on the **rising edge** of the blue pushbutton. That is, when the button is pressed. Signal edges graphically depicted:



Rising edges and falling edges, from Wikipedia article Signal edge

You can also listen for the **falling edge**: when the button is released:

```
GPIO.add_event_detect(BTN_B, GPIO.FALLING, lambda pin: GPIO.output(LED_B, True))
```

This does the opposite. Note that I am using active-low outputs, per the configuration previously detailed in *Raspberry Pi 3 GPIO: pushbuttons, LEDs for RC and BARR*, so this turns off the LED.

Both edges

Turn on the LED when the button is pressed, turn it off when released… would be nice to combine these two, something like this:

```
GPIO.add_event_detect(BTN_B, GPIO.RISING, lambda pin: GPIO.output(LED_B, False))
GPIO.add_event_detect(BTN_B, GPIO.FALLING, lambda pin: GPIO.output(LED_B, True))
```

but this fails:

```
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
RuntimeError: Conflicting edge detection already enabled for this GPIO channel
```

Turns out instead of specifying GPIO.RISING and GPIO.FALLING, there is another option: **GPIO.BOTH**, triggering on rising or falling.

OK, but then how do you know whether the edge was rising or falling? My solution was to read the input using GPIO.input() and act accordingly:

```
def handle(pin):
 GPIO.output(LED_B, not GPIO.input(BTN_B))

GPIO.add_event_detect(BTN_B, GPIO.BOTH, handle)
```

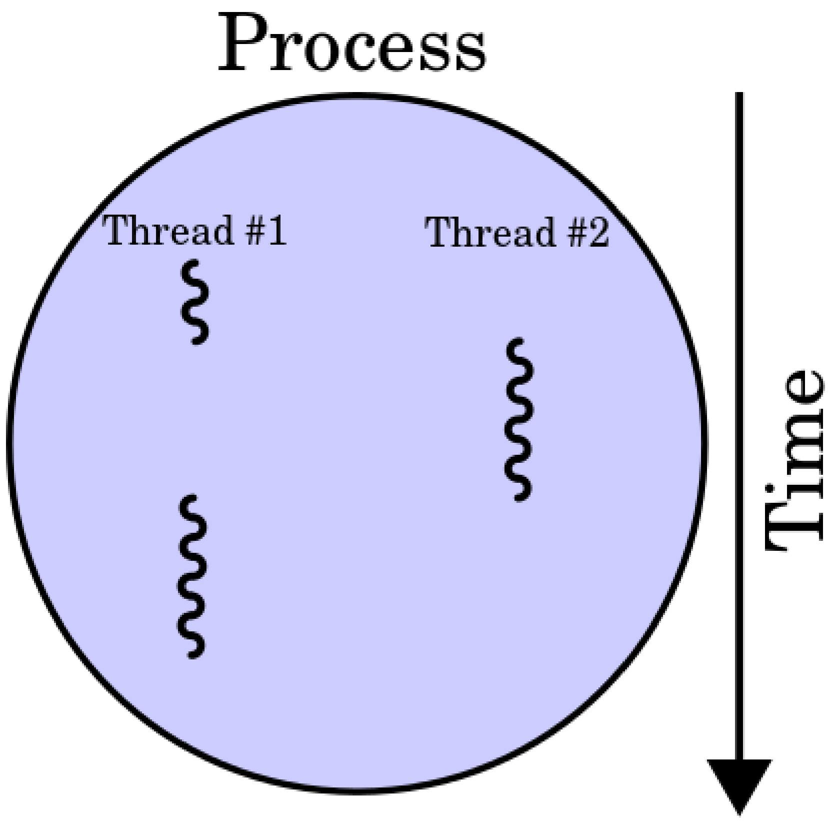
This is for blue, but can be trivially extended to the yellow, red, and green LEDs/buttons.

Threaded blinking

Lighting up lights when buttons are pressed isn’t the most exciting use of GPIO, arguably GPIO is even overkill for this purpose because one could simply wire up the LEDs directly to the pushbuttons, obviating the need for a Raspberry Pi at all. But remember, this is merely an example—nonetheless, let’s do something slightly more interesting.

gptest.py had an easter egg where if both red and green buttons are pressed simultaneously, the script would enter blink mode, and toggle all of the outputs on each iteration, at a rate controllable by yellow and blue buttons. This rate corresponded to the polling interval, but now that polling is gone, how can we cycle through blinking of the LEDs?

One solution: **threading**.



Two threads over time, diagram from Wikipedia article [Thread (computing)]

When the special button combination is detected, kick off a new thread with a loop to blink the LEDs. The same loop also reads the button inputs to control the blinking rate, and/or terminate the thread.

Putting it all together:

/home/pi/gpio/gpint.py:

Daemon & interactions

Now that the new super-efficient interrupt-driven gpint.py script is done, it can be executed as a background daemon, running all the time ready to read pushbutton inputs and light up LEDs as requested.

Note that other software may also use the same LEDs. How does gpint.py interact with this other software? Since the inputs in gpint.py are edge-triggered, flightled (etc.) can light up the LEDs itself, with no conflict with gpint.py most of the time. If a user presses the buttons, then the LED state will be overridden. This can be used both to force an LED on (by pressing and holding), or to force it off (by pressing and releasing).

Anyways without further ado, here is the init script and setup commands:

```
chmod a+x /home/pi/gpio/gpint.py
sudo ln -s /home/pi/gpio/gpint /etc/init.d
sudo /etc/init.d/gpint start
sudo ln -s /etc/init.d/gpint /etc/rc5.d/S01gpint
```

/etc/init.d/gpint:

- Raspberry Pi
- Technology
- Gpio
- Led
- Interrupt