Embedded Development

Working with Raspberry Pi

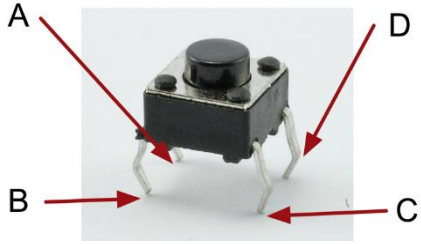
SAPPORO IT FRONT

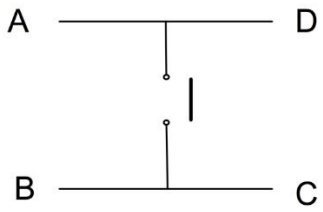
Rules

- You might actually damage hardware!
  - Double check when you do wiring
  - Do not connect any unintended GPIO pins, or you will short circuit the Pi
  - During wiring, Power off the Pi
  - Do not touch with wet fingers
  - OS on SD is vulnerable. Use proper sequence to operate
    "sudo shutdown –h now"
    "sudo reboot"
- Think "why", and consult hardware reference for better understanding
  - Reading hardware references is a must-have skill
  - You don't have to read a reference thoroughly at the beginning (Later, you will)

Detecting Switch State

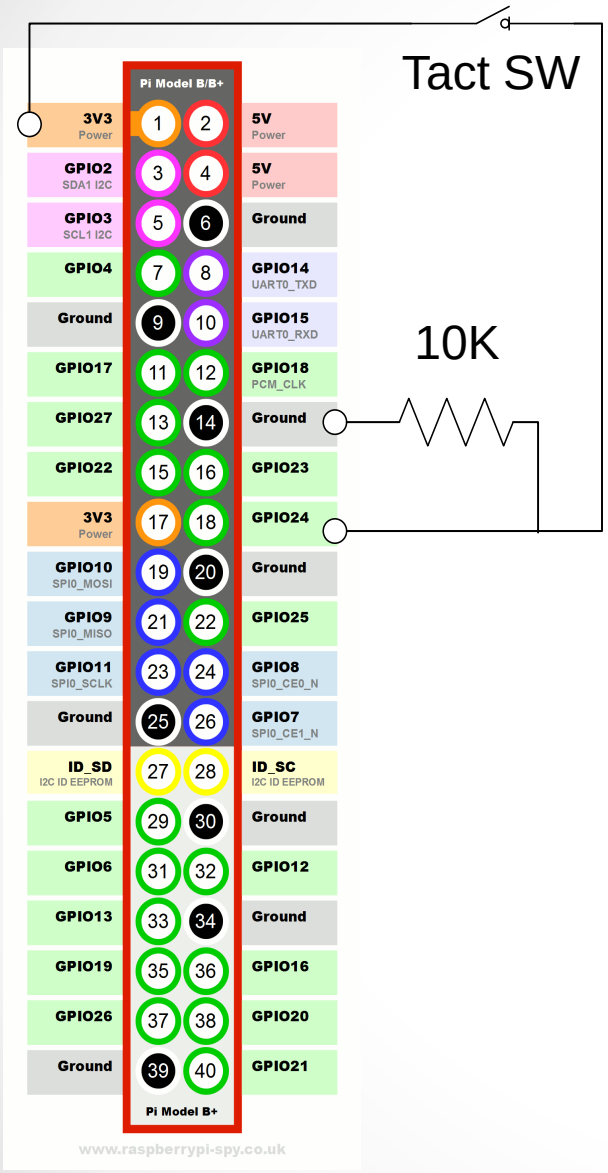
- Let's programmatically detect input from the device
- We use tactile (tact) switches as input devices





- Be very aware of its shape!
  - A-D, and B-C are connected, but nether A-B nor D-C
  - Connect A or D to VDD
  - connect B or C to GND
  - When you press a button, the circuit will be connected (current goes through)

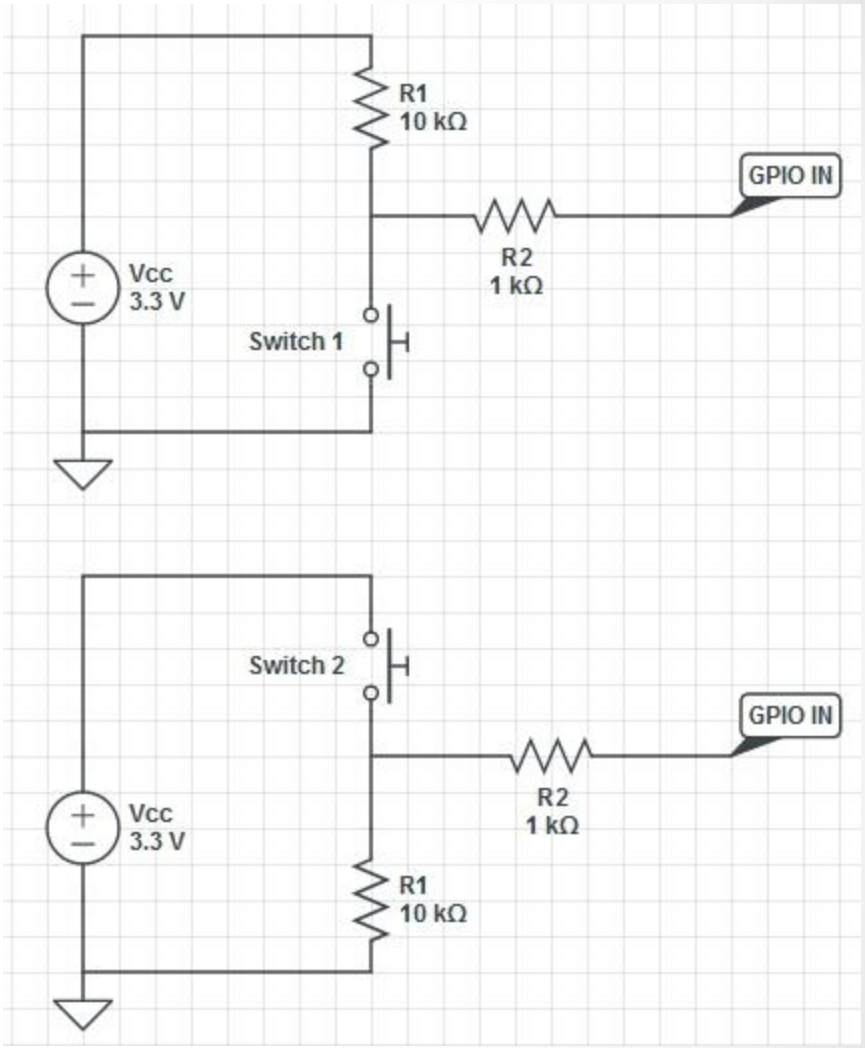
Switch - circuit



- When you press down the switch, GPIO24 (Pin18) get sink current
- Other wise, GPIO24 is connected to GND
- The role of resistor, like this usage, is called "pulldown resistor"
  - The 10K resistor ensures that only a little current is drawn when the switch is pressed
  - By using the pulldown, you make sure GPIO24 reads LOW, and when pressing the switch GPIO24 reads HIGH
- Consider without the resistor and the state SW open
  - The circuit is not closed
  - **You cannot tell HIGH or LOW on GPIO24!**

Pull-up and Pull-down

- Pull-up resistor



- Pull-down resistor

Switch - Software

- So, now you know what to do with your software

1. Set GPIO24 (pin18) to input
2. Inside a loop, read value of the pin
   - If it reads 1, the button is being pressed
   - If it reads 0, the button is not pressed

- Let's implement

1. Boot up your Pi
2. Connect via Remote Desktop
3. "sudo idle" to start IDLE with root priviledge
4. "CTRL+n" to create a new file

5. Type the following (be careful with 4-spaces indent), and save as "switch.py"

```
import RPi.GPIO as GPIO

GPIO.setmode(GPIO.BCM)
GPIO.setup(24, GPIO.IN, pull_up_down = GPIO.PUD_UP)

while True:
    if(GPIO.input(24) == 1):
        print("Button pressed")
GPIO.cleanup()
```

6. Hit F5 to run the program
7. Press the tactile switch and observe the console output
8. CTRL+C to exit the program

The program will cause some problems later, because it keeps running at full-speed and occupying the CPU resources. In an actual case, you should use sleep() at proper places.

- The following code lowers the sampling rate for the switch state detection, but more comfortably works with other processes inside OS

```
import RPi.GPIO as GPIO
Import sleep from time

GPIO.setmode(GPIO.BCM)
GPIO.setup(24, GPIO.IN, pull_up_down = GPIO.PUD_UP)

while True:
    time.sleep(0.01)  # wait 10 ms to give CPU chance to do other things
    if(GPIO.input(24) == 1):
        print("Button pressed")
GPIO.cleanup()
```

- The further better solutions would be to use an **interrupt**

Interrupts

- With interrupts, you can define a certain condition to watch for, and define what to do the condition occurs
- For the given condition, the defined what-to-do ("event handler", or "callback", or "interrupt service routine") will kick in whatever the process is currently doing.
- **Internally, the callbacks runs in a separate dedicated thread for callback,** which means that callback functions can be run at the same time as your main program, in immediate response to an edge.
- Be aware, multiple callbacks still in a same thread, so the callbacks run in sequence
- **When working for embedded software, utilizing interrupts is a key to success**
- One of the utilization is a design pattern called "State Machine"

- Using Rpi.GPIO, you can set the condition and the callback

```
GPIO.setup(18, GPIO.IN, pull_up_down=GPIO.PUD_UP)
GPIO.add_event_detect(18, GPIO.FALLING, callback=my_callback)
```

the above example, Pin 18 is set as input and internal pull-up resistor is activated, so default value is HIGH.
When the event of Pin18 start falling, the function named "my_callback" will be executed, even the process calls sleep()

- This way, you can remove the code for polling current value of a given pin!

Hardware Gotchas and workarounds

- Sometimes, the callbacks are called more than once for each button press.
- This is as a result of what is known as "switch bounce", or "chattering". There are 3 ways of dealing with switch bounce
  - Add a 0.1uF capacitor across your switch.
  - Software debouncing
  - A combination of both
- Software debouncing is easy with RPi.GPIO. Just specify the duration for ignoring repeated event occurrences

```
GPIO.add_event_detect(18, GPIO.FALLING, callback=my_callback, bouncetime=200)
```

this ignores 200ms for GPIO.RISING event

- Hardware engineers put efforts to reduce those hardware gotchas normally, especially if it's a product hardware (and budget allows)

Detecting Switch Input by using Interrupts

- Implement a new program which detects a switch state using the interrupts and a callback, based on the circuit on slide no. 4 and source on slide no.8

- Inside main process, do infinite count up by each second

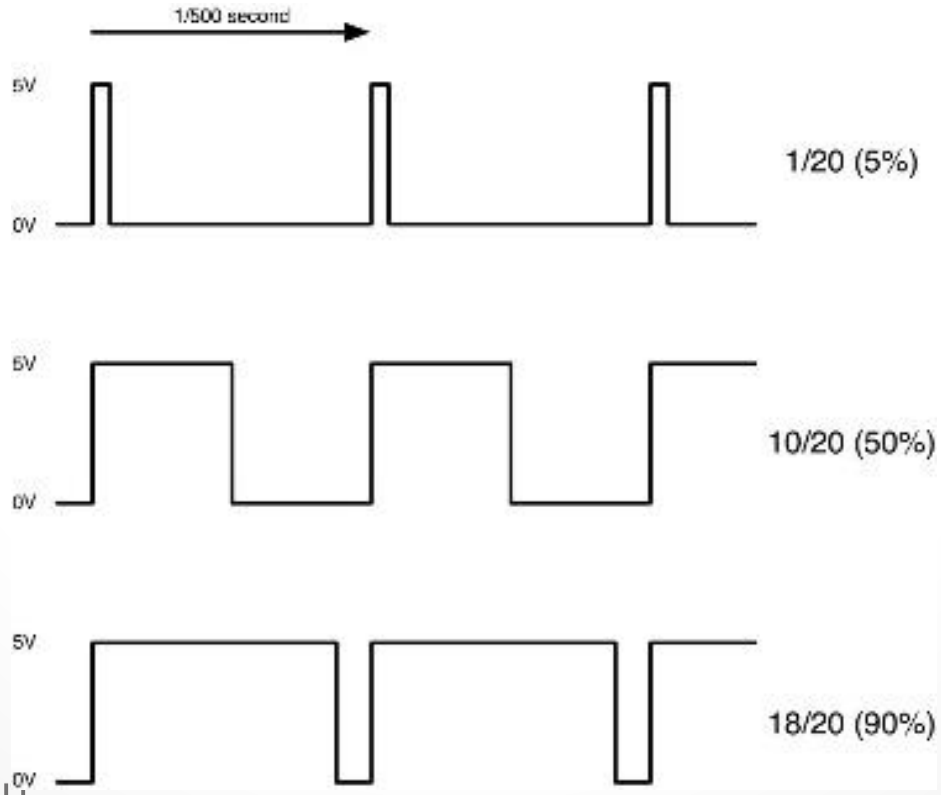
```
i = 0
while True:
    i = i + 1
    print(i)
    time.sleep(1)
```

- Inside the callback, just print "the button pressed"

Controlling Voltage Using Digital Output - PWM

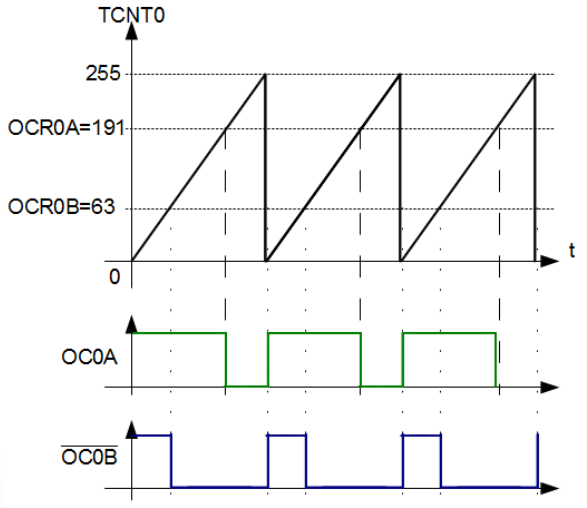
- Currently, we change the state of GPIO either HIGH or LOW. It's digital
- At HIGH, you output 3.3V, and at LOW, ground level voltage
- What if you want to change 1.65V?

- Use Pulse Width Modulation (PWM)
- At every given interval, you switch on and off the 3.3V output



- The ratio of length of HIGH against the interval is called as "duty-ratio"
- If you set 50% duty-ratio, the output voltage becomes 16.5V

- In old days (on old devices), you have to create the signal by combining a timer and a digital output
  - You start a timer at a given interval, and set HIGH for the output pin
  - The timer start counting up
  - At the threshold value, you set LOW for the output pin
  - By changing threshold, you can change the output voltage
  - When the timer reaches the maximum count and reset, you put HIGH again

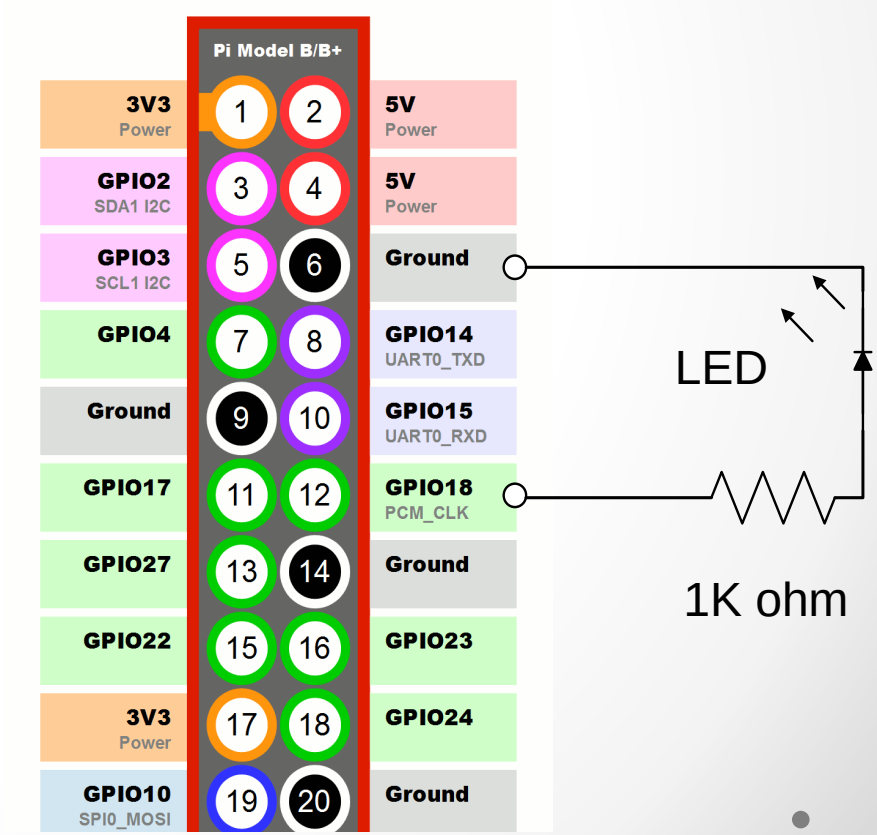


- Now modern devices has PWM mode, so you don't have to create the signal by your own

- There is one hardware PWM output, which is exposed as Pin12 (GPIO18). (If you need more PWM outputs, use I2C or SPI interface)

Let's control brightness of a LED.

1. Wire as the right diagram



2. Use IDLE, write the following code, and save as "brightness_control.py"

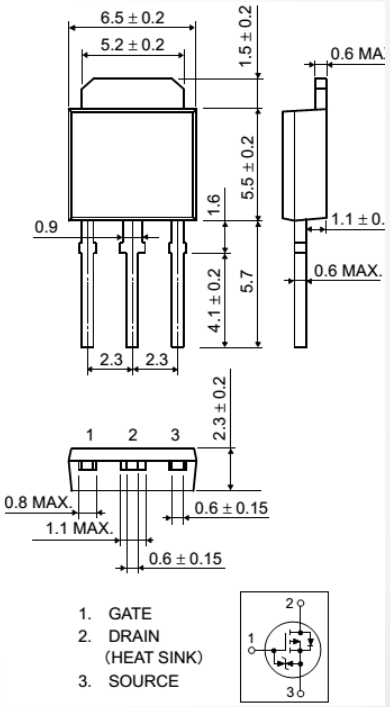
```
import RPi.GPIO as GPIO

GPIO.setmode(GPIO.BCM)
GPIO.setup(18, GPIO.OUT)
pwm_led = GPIO.PWM(18, 500)
pwm_led.start(100)
while True:
    duty_s = raw_input("Enter Brightness (0 to 100):")
    duty = int(duty_s)
    pwm_led.ChangeDutyCycle(duty)
```

3. Run the program, specify the duty ratio and observe the brightness changes of LED

Controlling High-Voltage Peripherals

- Now you know how to control the voltage output using PWM
- However, the Pi can only provide 3.3V or 5V from itself
- When you want to control a motor or devices which requires higher voltage, such as 9V or 12V, you need an extra power supply in order to drive them
- Furthermore, even for the 5V device, a device like a motor requires much power to drive it. If you supply 5V to the motor from the Pi, system will become unstable.
  So adding dedicated power supply for those is always a good idea
- There are 2 solutions:
  - Buy and use a dedicated integrated IC
  - Add a transistor called as MOSFET (metal oxide semiconductor field effect transistor)

- We use MOSFET "2SJ681" to control a motor, which drive with 5V
- With MOSFET, when you provide a certain voltage to a gate pin, circuit will connect from a source pin to a drain pin
  (think the gate as a switch)
- Please make sure you understand the pins' role
  (gate – drain - source) from the right diagram.
  The Iron plate is a good marker for the identification of pins
- Connecting 5V to source, and apply 3.3V on gate,
  then you will get 5V output voltage from drain
- Make sure ground level for both 3.3V and 5V circuits are equalized (connect both GND)
- Technically speaking 2SJ681 is 4V gate drive, but it works with 3.3V



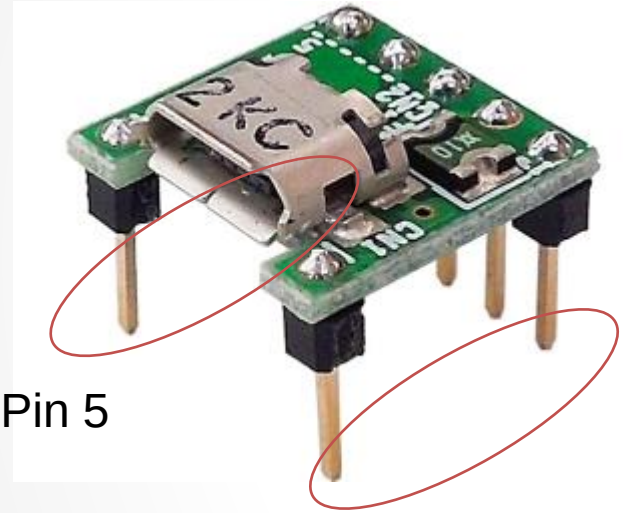
- 2SJ681 is a **P-channel Enhancement Mode MOSFET**

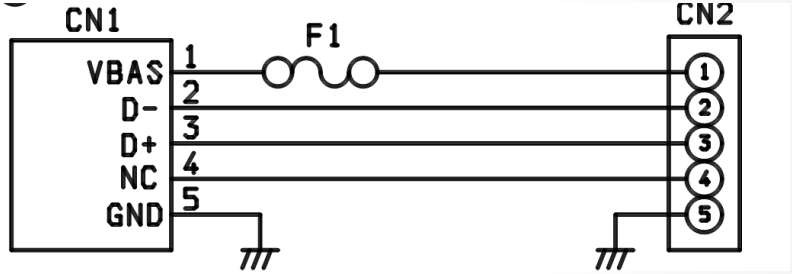
- There are two types: N-Channel (NMOS) or P-Channel (PMOS)
  - N-Channel
    the source is connected to ground. To turn the MOSFET on, we need to raise the voltage on the gate. To turn it off we need to connect the gate to ground.

  - P-Channel
    The source is connected to the power rail (Vcc). In order to allow current to flow the Gate needs to be pulled to ground. To turn it off the gate needs to be pulled to Vcc.

- In short:
  To connect a source and a drain, you set HIGH for a pin connected to the gate
  To disconnect a source and a drain, you set LOW

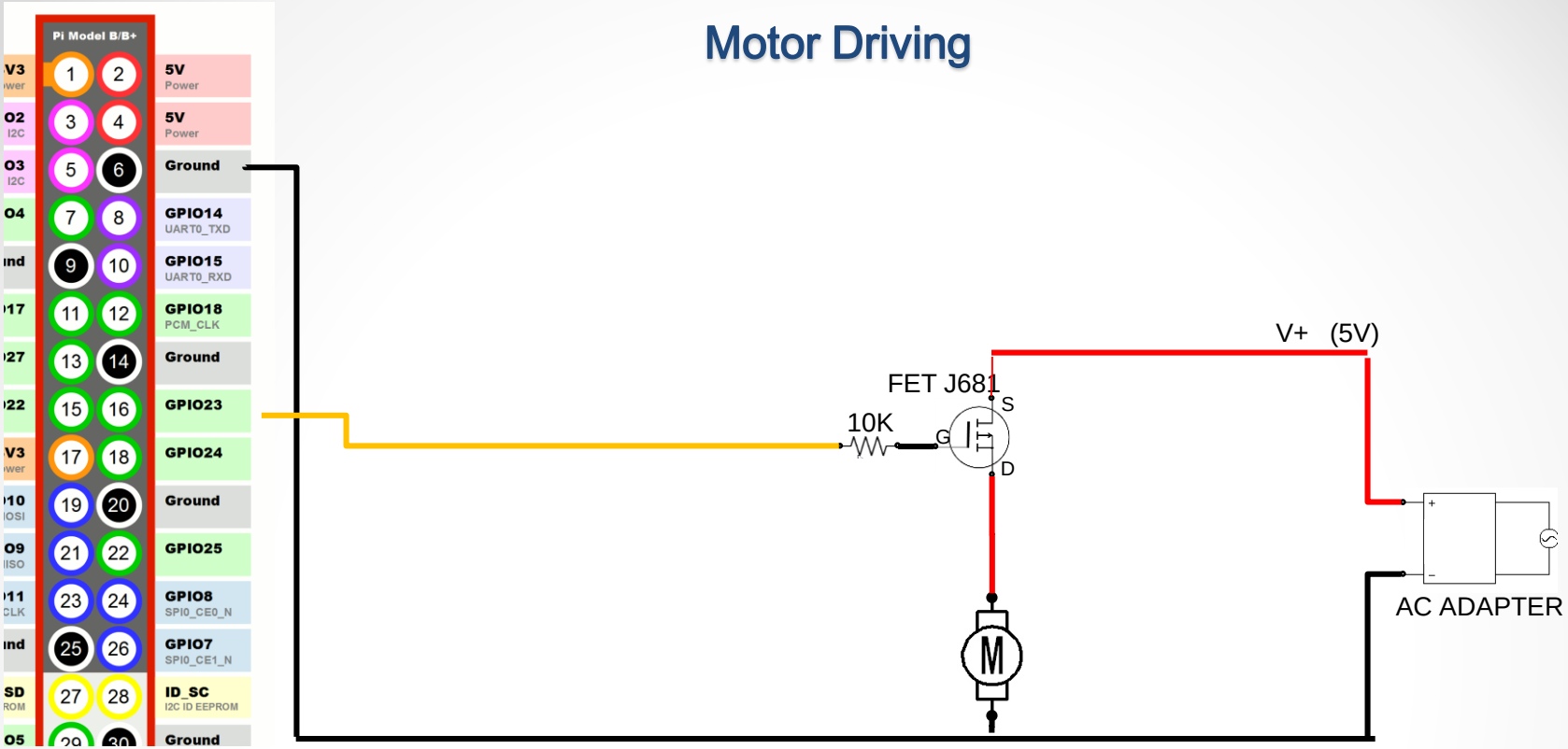
- For 5V power supply, we use: AE-USB-MICRO-B-D
  - Connect to micro USB, which can supply over 1A
  - You can get 5V from Pin 1, GND from Pin 5 for both channels





- The reason we supply 5V from Pin 2 or 4 of the Pi, is if you draw a lot of current from them, your Pi becomes unstable (reboot, freeze, unstable program execution, etc)

Motor Driving



- **Make sure you use a vise to fix the motor on table surface,** or you will hurt yourself

- Software is rather simple. Just set GPIO23 (Pin:16) to output direction, and on and off in every 5 seconds. Name the program as "motor_drive.py" (Yes, your assignment. Do it now!)

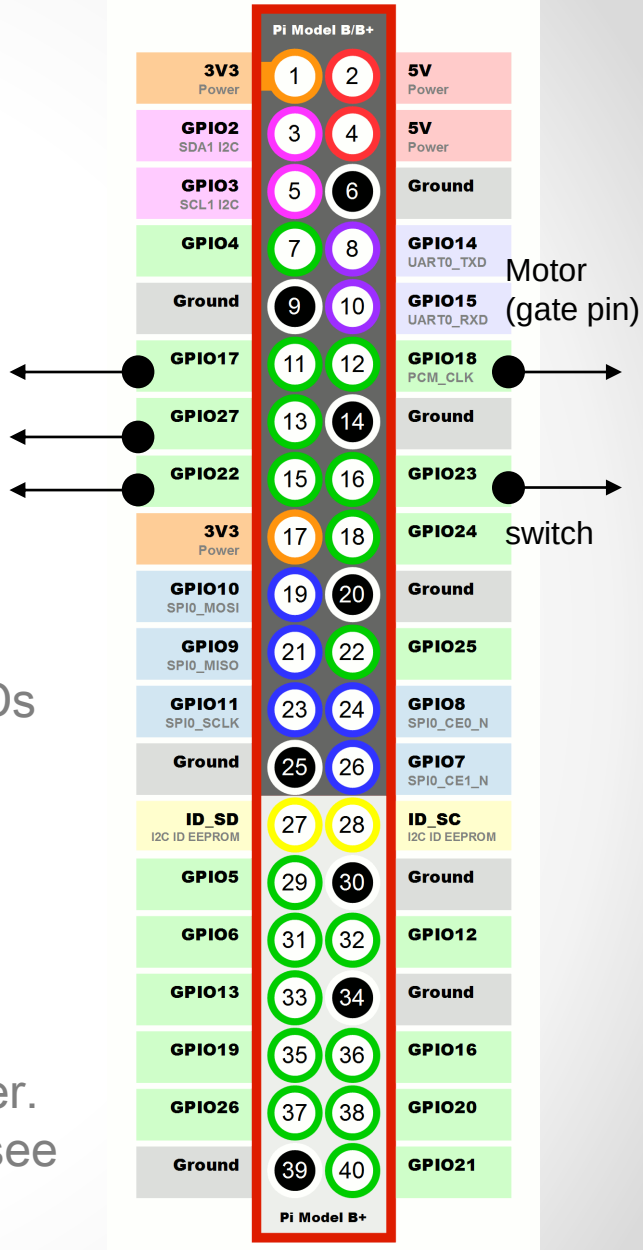
I/O integration

- Let's combine input and output
- Using bullet boards:
  - Wire 3 LEDs
  - Wire 1 switch
  - Wire a motor

- By pressing the switch, start motor
- By pressing the switch again, stop motor
  - Use interrupts and callbacks
- While motor running, do the Knight rider with 3 LEDs

- You have to hold current state of motor pin, using a global variable, in order to toggle the behavior of the switch

- Main program do the infinite loop for the Knight rider. You check the global variable at every iteration to see the current motor state.



```
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BOARD)
print "---------- init ----------"
print "setting Pin:16 to INPUT and pulled-up"
GPIO.setup(16, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
time.sleep(5)

print "setting Pin:16 to OUTPUT and HIGH"
GPIO.setup(16, GPIO.OUT, GPIO.HIGH)
time.sleep(5)
print "--------------------------"

print "LOW will start the motor"
GPIO.output(16, GPIO.LOW)
time.sleep(5)

print "HIGH will stop the motor"
GPIO.output(16, GPIO.HIGH)
time.sleep(5)

print "LOW will start the motor"
GPIO.output(16, GPIO.LOW)
time.sleep(5)

print "HIGH will stop the motor"
GPIO.output(16, GPIO.HIGH)
time.sleep(5)

print "LOW will start the motor"
GPIO.output(16, GPIO.LOW)
time.sleep(5)

print "now stop"
GPIO.output(16, GPIO.HIGH)

time.sleep(3)

GPIO.cleanup()
print "---------- exit ----------"
```