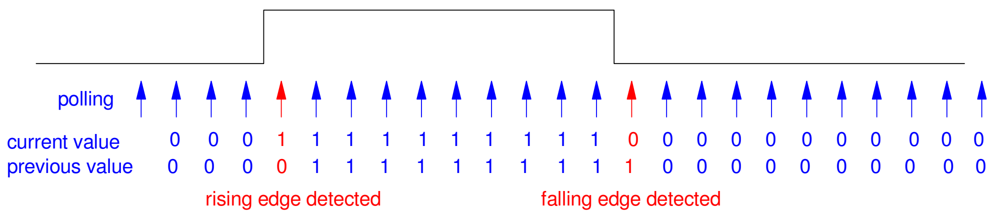
16. Edge Interrupts

Introduction:

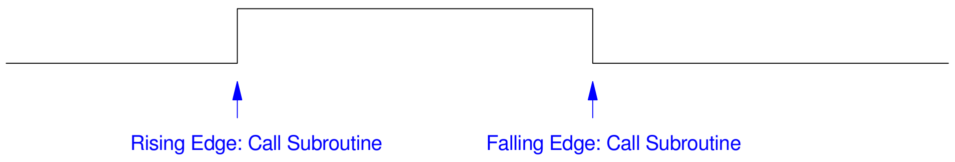
Interrupts are a strange beast - but they're really useful. An interrupt is a subroutine called by hardware. In this lecture, the subroutine is called by a rising or a falling edge (or both) on a given pin. What makes interrupts strange is that, with them, subroutines are being called in your program by someone other than you. What makes them really useful is they allow you to take action without having to constantly checking a pins status over and over.

Currently, we've been using polling to detect rising and falling edges of a pin (or to detect when you press a button.) Polling is really inefficient: you have to check the status of a pin over and over again.



polling: check the status over and over so you detect the rising or falling edge

With interrupts, you can instead simply call a subroutine only when the edge is detected. This makes the code much more efficient - albeit a bit more confusing.



Interrupts: Only call the subroutine when you detect the falling edge (detected by hardware)

Because interrupts are called by someone else, passing data to and from interrupts almost has to happen through global variables. This makes Advanced Embedded a little different from Computer Science I. In the latter course, you're told *never never use global variables.* In this course, we ignore that and have no problem with using global variables - mainly because there's no other option. Everyone can see global variables: subroutines you call can see them. Subroutines you didn't call can see them (interrupts).

Edge Interrupts: Up Counter

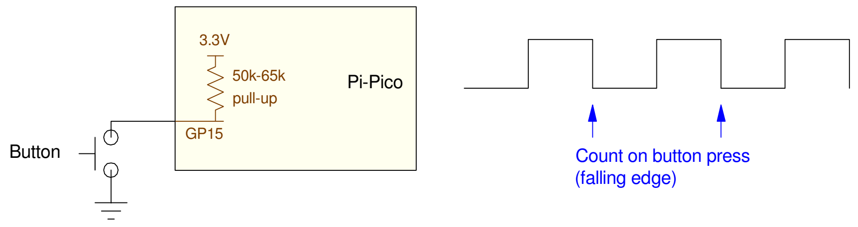
To give an idea of how edge interrupts can be used, let's create an up-counter:

- Increase a count by one each time you press button GP15.

Similar to before, assume

- GP15 is connected to an internal pull-up resistor to +3.3V, and
- When you press the button, pin GP15 is grounded.

This results in a button-press corresponding to a falling edge on the signal on GP15



Up-Counter: Count every button press by triggering an edge interrupt of the falling edges on GP15

In Python, you can trigger an edge interrupt with the following code:

- (line 3&4): Global variables are used to pass data from the main routine and the interrupt
- (line 6): GP15 is set to input with an internal pull-up resistor (line 6)
- (line 7): The interrupt service routine (any name works)
- (line 13): Set up an interrupt for a falling edge, calling subroutine *IntServe*

```
1    # Up Counter using interrupts
2    from machine import Pin
3
4    interrupt_flag=0
5    N = 0
6
7    pin = Pin(15,Pin.IN,Pin.PULL_UP)
8    def IntServe(pin):
9        global interrupt_flag
10       global N
11       interrupt_flag=1
12       N = N + 1
13
14   pin.irq(trigger=Pin.IRQ_FALLING, handler=IntServe)
15
16   while(1):
17       if(interrupt_flag):
18           print("N = ", N)
             interrupt_flag=0
```

Shell

```
N = 1
N = 2
N = 3
```

Note the following:

- Each time you press the button, an interrupt is called (falling edge)
- Data is passed using global variables. That's about the only way to share data with subroutines you didn't call (hardware called the subroutine)
- If you want to detect rising edges instead (i.e. button releases), change line 13 to

```
        pin.irq(trigger=Pin.IRQ_RISING, handler=IntServe)
```

- If you want to count *both* rising and falling edges, change line 13 to

```
        pin.irq(trigger=Pin.IRQ_RISING | Pin.IRQ_FALLING, handler=IntServe)
```

Also note that interrupt service routines are very simple: you get in and then get out quickly as possible. This is necessary since you don't know when the next interrupt is coming.

As a general rule:

- Don't use loops inside interrupts: they take too long to execute.
- Never use a while-loop inside an interrupt: if you're stuck inside the interrupt the main routine doesn't execute.
- If-statements are OK - you only evaluate these once.
- Any variables you want to save for later or pass to the main routine need to be global variables.

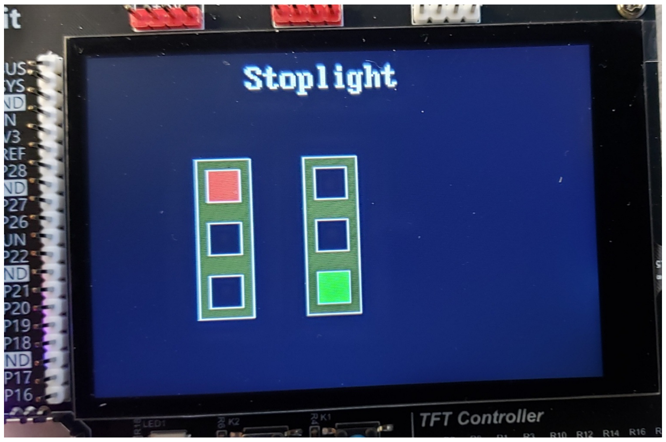
StopLight & Bottom-Up Programming:

As a second example of using edge interrupts, let's use edge interrupts to program a stoplight with a N/S light (left) and an E/W light (right). Rather than writing the entire program all at once, let's use subroutines and bottom-up programming.

Bottom-up programming is one technique for writing and testing your code step-by-step. It's a useful technique for larger programs that reduces the frustration you get when your code refuses to compile and run.

Starting out, let's write the display routine. The reason for starting here is having a visual output helps you figure out why different sections of your code is failing. Start with a subroutine which is passed

- The status of the light (0=red, 1=yellow, 2=green), and
- The (x,y) location of the light on the LCD screen



Stoplight Program: Display an E/W and N/S stoplight on the LCD display

A subroutine which does this is as follows.

```
def Display(Light, x, y):
    Red = LCD.RGB(200,0,0)
    Yellow = LCD.RGB(200,200,0)
    Green = LCD.RGB(0,200,0)
    Brown = LCD.RGB(30,30,0)
    Black = 0
    White = LCD.RGB(200,200,200)

    LCD.Solid_Box(x, y, x+50, y+150, Brown)
    LCD.Box(x, y, x+50, y+150, White)

    y += 10
    x += 10
    if(Light == 0):
        LCD.Solid_Box(x,y,x+30,y+30,Red)
    else:
        LCD.Solid_Box(x,y,x+30,y+30,Black)
    LCD.Box(x,y,x+30,y+30,White)

    y += 50
    if(Light == 1):
        LCD.Solid_Box(x,y,x+30,y+30,Yellow)
    else:
        LCD.Solid_Box(x,y,x+30,y+30,Black)
    LCD.Box(x,y,x+30,y+30,White)

    y += 50
    if(Light == 2):
        LCD.Solid_Box(x,y,x+30,y+30,Green)
    else:
        LCD.Solid_Box(x,y,x+30,y+30,Black)
    LCD.Box(x,y,x+30,y+30,White)
```

To test this subroutine, you can run some test code:

```
# Test code for stoplight
import time
import LCD

def Display(Light, x, y)
    :

Navy = LCD.RGB(0,0,5)
White = LCD.RGB(200,200,200)
LCD_Init()
LCD_Clear(Navy)
LCD.Text2('Stoplight',200,20,White,Navy)
Display(0, 100, 100)
Display(2, 200, 100)
```

This results in the previous image.

Step 2: Once the display routine works, add a main routine which cycles through the green/yellow/red sequence, changing every second:

N (counter)	0	1	2	3	4	5
E/W	Green	Yellow	Red	Red	Red	Red
N/S	Red	Red	Red	Green	Yellow	Red

Simply counting mod six with a 1-second wait at the end works:

```
# Main Routine: Cycle lights every 1.00 second
Red = 0
Yellow = 1
Green = 2
while (1):
    N = (N + 1) % 6
    if(N == 0):
        Display(Green, 100, 100)
        Display(Red, 200, 100)
    if(N==1):
        Display(Yellow, 100, 100)
        Display(Red, 200, 100)
    if(N==2):
        Display(Red, 100, 100)
        Display(Red, 200, 100)
    if(N==3):
        Display(Red, 100, 100)
        Display(Green, 200, 100)
    if(N==4):
        Display(Red, 100, 100)
        Display(Yellow, 200, 100)
    if(N==5):
        Display(Red, 100, 100)
        Display(Red, 200, 100)
    print(N)
    time.sleep(1)
```

Note: While this routine works, it's really inefficient. >99% of the time is spent in the sleep() statement.

Step 3: Once this works, add an interrupt to count rising edges of an external clock (or a person pressing a button). This greatly increases the efficiency of the program. Rather than waiting in a sleep() statement, the main routine only enters a if-statement when needed.

When you press a button and trigger an interrupt

- A counter increments (N counts mod 6), and
- A flag is set, telling the main routine that the status of the light changed

This allows the main routine to do something else (which in this case is nothing) - and only take action when the light changes.

```
# Interrupt Service Routine
from machine import Pin
import time

N = 0
flag = 0

pin1 = Pin(15,Pin.IN,Pin.PULL_UP)

def CLK(pin1):
    global N, flag
    flag = 1
    N = (N+1)%6

pin.irq(trigger=Pin.IRQ_FALLING, handler=CLK)
```

Interrupt Service Routine: Counts button presses and sets a flag when a button is pressed

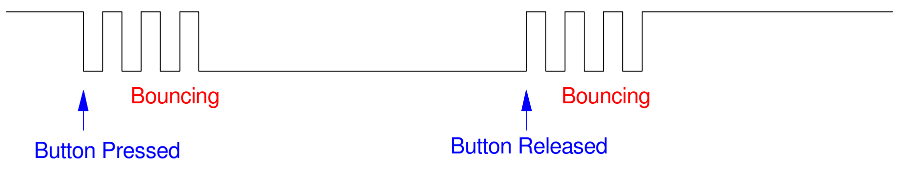
```
# Main Routine
# (only enters the display routine when the light changes)
while (1):
    if(flag == 1):
        flag = 0
        if(N == 0):
            Display(Green, 100, 100)
            Display(Red, 200, 100)
        if(N==1):
            Display(Yellow, 100, 100)
            Display(Red, 200, 100)
        if(N==2):
            Display(Red, 100, 100)
            Display(Red, 200, 100)
        if(N==3):
            Display(Red, 100, 100)
            Display(Green, 200, 100)
        if(N==4):
            Display(Red, 100, 100)
            Display(Yellow, 200, 100)
        if(N==5):
            Display(Red, 100, 100)
            Display(Red, 200, 100)
        print(N)
```

Main Loop: The LCD is updated only when needed - leaving the main routine time to do other stuff (nothing in this case)

Some problems with this routine is

- You have to manually press for each change in the light. We'll fix this when we cover timer interrupts in the next lecture.
- Sometimes you get multiple counts due to bouncing in the button.

Debouncing: Bouncing is when a mechanical switch opens and closes several times each time you press or release the button.



Bouncing is when you get multiple edges each time you press and release a button

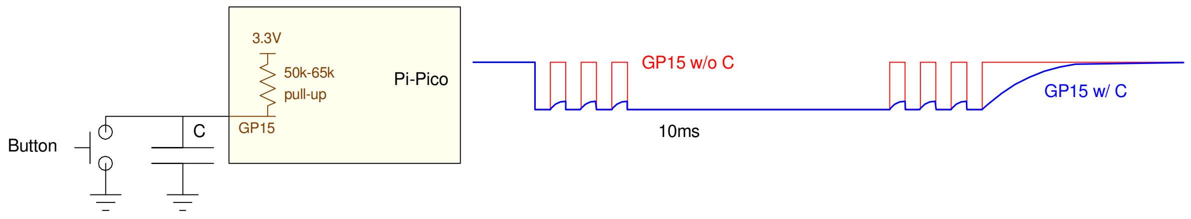
In order to eliminate bouncing, both hardware and software can be used.

Hardware Debouncing: By adding a capacitor, you can filter out the bouncing. Picking C so that the RC time constant is equal to the bounce time usually works. Assuming this is 10ms:

$$RC = 10ms$$

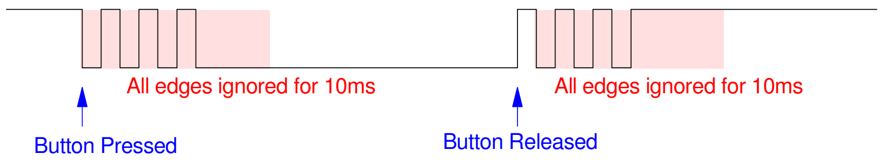
R is the internal pull-up resistor (50k to 65k), resulting in

$$C = 153nF..200nF$$



Adding a capacitor filters is one way to remove bouncing

Software Debouncing: Anything you can do in hardware you can do in software. In software, you can add a dead-time: once you detect an edge, all subsequent edges within 10ms are ignored.



Software Debouncing: All edges are ignored for 10ms once a falling edge is detected

The following routine does this using a sleep() statement. This isn't the best programming practice, since it locks up the entire computer inside an interrupt for 10 ms. When we get to timer interrupts (next lecture), we'll be able to ignore edges for 10ms without locking up the processor. In addition, counts on the rising edge are avoided by checking the status of the I/O pin after 10ms:

- If the pin is low, you detected a falling edge (count)
- If the pin is high, you triggered off of bouncing during a rising edge (ignore)

In Python code, change the interrupt service routine to include a 10ms wait (ignore all edges for 10ms once an edge is detected).

```
def CLK(pin1):
    global N, flag
    time.sleep(0.01)
    if(pin1.value() == 0):
       flag = 1
       N = (N+1)%6
```

Multiple Interrupts: Hungry-Hungry Hippo (Take 1):

Next, let's look at multiple interrupts. Specifically, let's write a program to play *Hungry-Hungry Hippo*. Here, once the game starts, each player has 10 seconds to press their button as often as they can. The person who presses their button the most number of times wins.

Hungry-Hungry Hippo: Can you beat a husky?
https://youtu.be/9Owv0h8wz-I?feature=shared

Checking for multiple button presses is pretty easy. In terms of hardware, add a push button for each player:

- Player A: presses button attached to GP15
- Player B: presses button attached to GP14

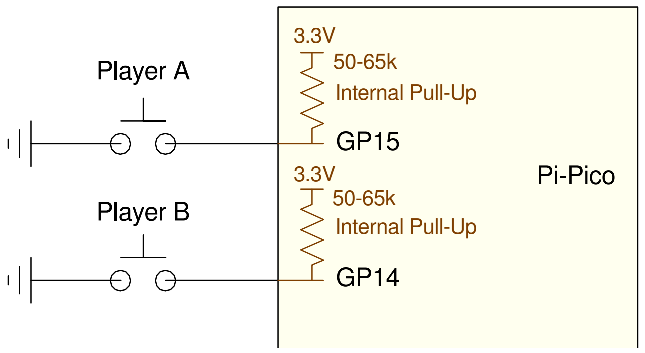
In terms of software,

- Add a counter for each player (these need to be global variables - N1 and N2 in the following program)
- Define the pins you're using to be input pins,
- Define a separate interrupt service routines for each input pin, and
- Set up each interrupt for a rising, falling, or both edges.

To set the game time to ten seconds, a while loop is used at the end along with a counter (Time).

- Each loop, Time is incremented by 0.1

- At the end of the main loop, the program sleeps for 100ms
- Once 10 seconds have elapsed, the program (and game) ends.



Hardware for Huntry-Hungry Hippo.
The push-button are already connected to the Pi-Pico Breadboard Kit
(no additional hardware needed)

```
# Hungry-Hungry Hippo (Take 1)

from machine import Pin
import time

N1 = 0
N2 = 0

pin1 = Pin(15,Pin.IN,Pin.PULL_UP)
pin2 = Pin(14,Pin.IN,Pin.PULL_UP)

def player1(pin1):
    global N1
    N1 = N1 + 1

def player2(pin2):
    global N2
    N2 = N2 + 1

pin1.irq(trigger=Pin.IRQ_FALLING, handler=player1)
pin2.irq(trigger=Pin.IRQ_FALLING, handler=player2)

Time = 0.0

while (Time < 10):
    print(Time, N1, N2)
    Time += 0.1
    time.sleep(0.1)
```

Software for Hungry-Hungry Hippo. Interrupts keep track of edges on GP14 and GP15

Note that the timing in this program is a little off - meaning each game is slightly longer than 10.00 seconds. This is due to the time it takes to execute the other statements in the main loop.

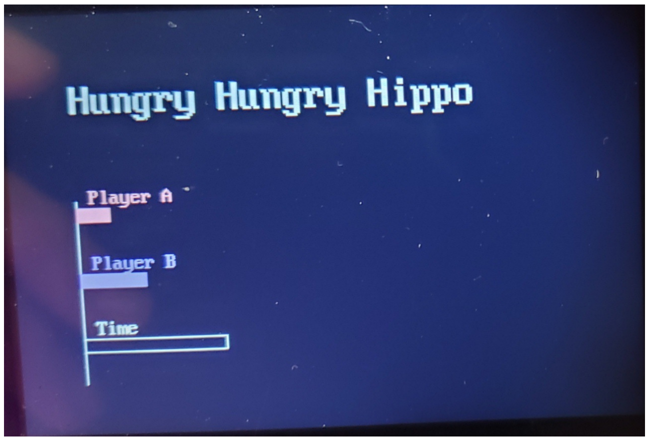
- Ideally, the main loop should execute every 100ms

- Actually, the execution time is slightly longer than 100ms due to the execution time of the *print* statement and the time it takes to do a floating-point addition.

This problem can be overcome by using *timer* interrupts (next lecture).

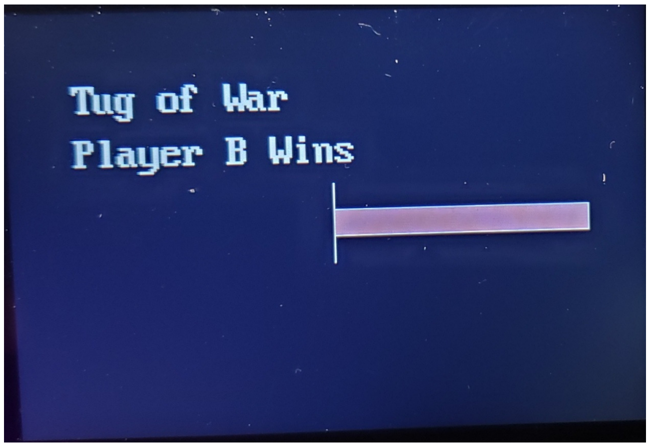
Once you are able to record the number of button presses for two (or more) players, you can vary the game with slight modifications along with the LCD display:

Hungry-Hungry Hippo (take 3): Play for a fixed amount of time. Display each player's score and the time remaining.



Variant #1: Display each player's score and the time remaining

Tug of War: Keep playing until one player's score is 30 more than the other's. Display the difference in score.



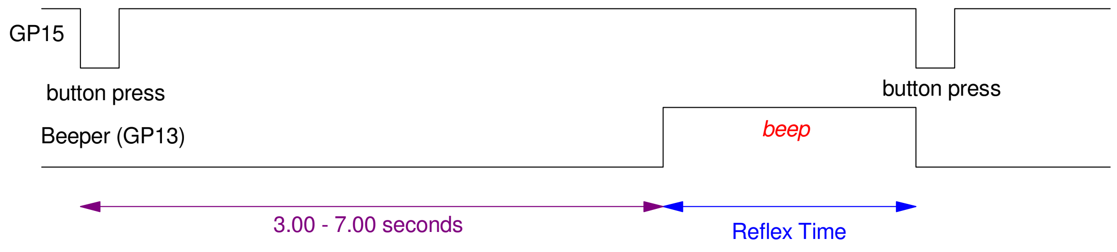
Variant #2: Display the difference in score. Play until one player is up 30 points

Reflex Game:

With edge interrupts you can record the time of an event (within a few clocks - the time it takes to trigger the interrupt). With this, you can measure your reflex time.

In the following program,

- The game starts by pressing a button connected to pin GP15.
- Once pressed, the Pico waits between four and seven seconds.
- Once done waiting, the beeper turns on and that time is saved
- The program then waits for you to press GP15 again
- Once pressed, the time between the beeper going on and pressing GP15 is recorded and displayed



Reflex Game: Press GP15 to start the game. Press GP15 as soon as you hear the beeper

One advantage of using edge interrupts is it's harder to cheat.

- If you are using the level or GP15, you can simply hold down the button. As soon as the beeper turns on, it will see GP15 being low (pressed).
- By detecting edges, this strategy won't work: the program only recognized high-to-low transitions (button presses.)

Sample code is on the following page. Note that by using interrupts, you're getting a very precise measurements: one part in 130,000 or so. That's a common trait of computers: they are *very* good at measuring time.

```
reflex game
from machine import Pin
import time
import random

beeper = Pin(13, Pin.OUT)
pin1 = Pin(15,Pin.IN,Pin.PULL_UP)

T0 = 0
T1 = 0
flag = 0

def Button15(pin1):
    global T1, flag
    T1 = time.ticks_us()
    flag = 1

# options are RISING, FALLING, LOW_LEVEL, HIGH_LEVEL
pin1.irq(trigger=Pin.IRQ_FALLING, handler=Button15)


while(1):
    beeper.value(0)
    flag = 0
    while(flag == 0):
        pass
    print('Starting:  4..7 seconds later')
    dT = random.random() * 4 + 3
    time.sleep(dT)
    flag = 0
    beeper.value(1)
    T0 = time.ticks_us()
    while(flag == 0):
        pass
    beeper.value(0)
    Reflex = T1 - T0
    print('Reflex Time(us) = ',T1-T0)
```

Shell

```
Reflex Time(us) =  274368
Reflex Time(us) =  162795
Reflex Time(us) =  141069
Reflex Time(us) =  133092
Reflex Time(us) =  140628
Reflex Time(us) =  139160
```

Optical Encoders & Pong Game

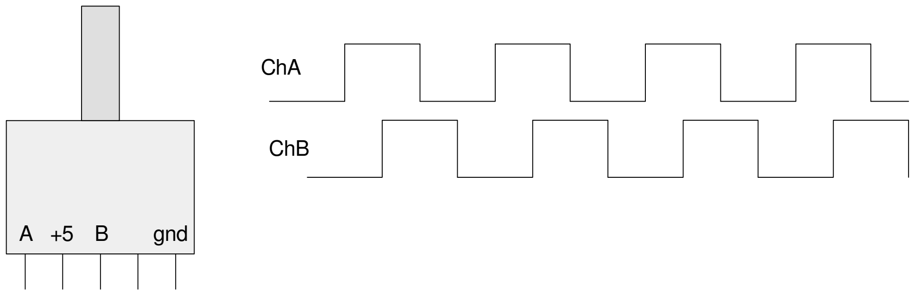
Finally, one of the more useful things you can do with edge interrupts is read an optical encoder - also known as a digital potentiometer.

Analog Potentiometers: A normal (analog) potantiometer usually consists of a long strand of resistance wire, such as shown in the figure below. If you connect the two ends of this resistor to 0V (left) and 3.3V (right), the wiper (or center tap) can pick off the voltage along this resistor, from 0V to 3.3V. The net results is an analog potentiometer allows you to output an analog voltage in the range of 0V to 3.3V

Analog Potentiometer: If you connect the ends of the resistor to 0V and 3.3V, the center tap can output any (analog) voltage in this range

By reading this votlage with an A/D, you can use an analog potentiometer to alter variables in your program. (As a sidelight, analog potentiometers typically have stops - preventing you from going past the endpoints (0% and 100% of input voltage).

Digital Potentiometers: In contrast, digital potentiometers typically outputs two squares waves in phase quadrature (i.e. 90 degrees apart).



Digital potentiometers output two square waves in phase quadrature

What this means is

- You can determine distance by counting the numbers of edges on Channel A, and
- You can determine direction by noting the value of channel B

The net result is, assuming an encoder with 100 pulses per rotation, you get 200 counts (200 edges on channel A) each rotation.

In Python, a endge interrupt on rising and falling edges on Channel A gives you these 200 counts per rotation.

```
from machine import Pin
import time

N = 0

pin1 = Pin(15,Pin.IN,Pin.PULL_UP)
pin2 = Pin(14,Pin.IN,Pin.PULL_UP)

def ChA(pin1):
    global N
    if(pin1.value() == pin2.value()):
        N -= 1
    else:
        N += 1

pin1.irq(trigger=Pin.IRQ_FALLING | Pin.IRQ_RISING, handler=ChA)

while (1):
    print(N)
    time.sleep_ms(100)
```

Python routine for counting rising and falling edges on Channel A

You can actually get double that by counting edges on both channel A and B. A little trial and error gives the following code (i.e. when rotating clockwise, both interrupt ChA and ChB increase N.)

```
from machine import Pin
import time

N = 0

pin1 = Pin(15,Pin.IN,Pin.PULL_UP)
pin2 = Pin(14,Pin.IN,Pin.PULL_UP)

def ChA(pin1):
    global N
    if(pin1.value() == pin2.value()):
        N -= 1
    else:
        N += 1

def ChB(pin2):
    global N
    if(pin1.value() == pin2.value()):
        N += 1
    else:
        N -= 1

pin1.irq(trigger=Pin.IRQ_FALLING | Pin.IRQ_RISING, handler=ChA)
pin2.irq(trigger=Pin.IRQ_FALLING | Pin.IRQ_RISING, handler=ChB)

while (1):
    print(N)
    time.sleep_ms(100)
```

Python routine for counting edges on both channel A and B

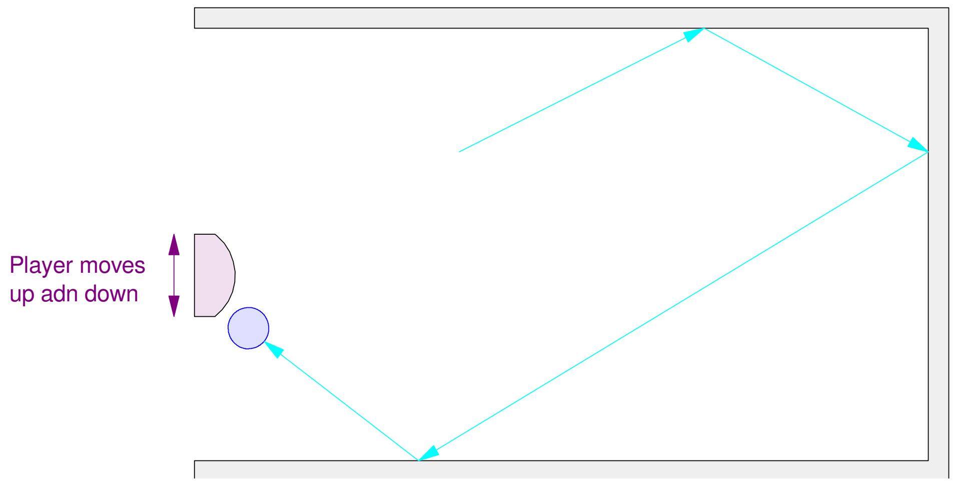
With this, you have a way to adjust parameters in your program by turning a knob.

Pong

As an example of how an optical encoder is useful, the following program recreates the arcade game *Pong*. In *pong*, a ball is bouncing around the screen.

- If the ball hits the top, right, or bottom wall, it bounces.
- If the ball hits the paddle on the left side of the screen, the ball bounces and the player gets one point.
- If the ball misses the padde and hits the left side of the screen, the game is over

The goal of *pong* is to move your paddle up and down to hit the ball, keeping it in play. Sort of like playing one-person ping-pong.



Pong Game: The player moves the paddle up and down to keep the ball in play

The program on Bison Academy works as follows:

- The position and velocity of the ball is specified at the beginning of each round.
- Numerical integration is used to update the ball position

$$x = \int v_x \cdot dt$$

$$y = \int v_y \cdot dt$$

- If the ball hits an edge, it bounces (the velocity switches sign).
- If the ball hits the left edge within 10 pixels of the paddle, it bounces and the ball remains in play
- If the ball hits the left edge more than 10 pixels away from the paddle, the round is over

In addition, the paddle is curved

- The y velocity is altered by how far away from the center of the paddle the ball hits

The main part of the program are the edge interrupts: these determine the y coordinate of the paddle.

```
from machine import Pin
import time

PaddleY = 0

pin1 = Pin(15,Pin.IN,Pin.PULL_UP)
pin2 = Pin(14,Pin.IN,Pin.PULL_UP)

def ChA(pin1):
    global PaddleY
    if(pin1.value() == pin2.value()):
        PaddleY += 1
    else:
        PaddleY -= 1

def ChB(pin2):
    global PaddleY
    if(pin1.value() == pin2.value()):
        PaddleY += 1
    else:
        PaddleY -= 1

pin1.irq(trigger=Pin.IRQ_FALLING | Pin.IRQ_RISING, handler=ChA)
pin2.irq(trigger=Pin.IRQ_FALLING | Pin.IRQ_RISING, handler=ChB)

while (1):
    print(PaddleY)
    time.sleep_ms(100)
```

Heart of Pong game: Encoders let the operator control the y-position of the paddle

References

Pi-Pico and MicroPython

- https://github.com/geeekpi/pico_breakboard_kit
- https://micropython.org/download/RPI_PICO/
- https://learn.pimoroni.com/article/getting-started-with-pico
- https://www.w3schools.com/python/default.asp
- https://docs.micropython.org/en/latest/pyboard/tutorial/index.html
- https://docs.micropython.org/en/latest/library/index.html
- https://www.fredscave.com/02-about.html

Pi-Pico Breadboard Kit

- https://wiki.52pi.com/index.php?title=EP-0172

Other

- https://docs.sunfounder.com/projects/sensorkit-v2-pi/en/latest/
- https://electrocredible.com/raspberry-pi-pico-external-interrupts-button-micropython/
- https://peppe8o.com/adding-external-modules-to-micropython-with-raspberry-pi-pico/
- https://randomnerdtutorials.com/projects-raspberry-pi-pico/
- https://randomnerdtutorials.com/projects-esp32-esp8266-micropython/