17. Timer Interrupts

Introduction:

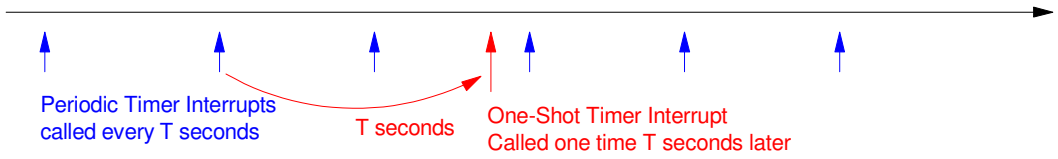
In the previous lecture, we looked at edge interrupts: subroutines which are called by hardware detecting a rising or a falling edge. In this lecture, we look at timer interrupts: subroutines which are called by hardware after a fixed amount of time has elapsed.

Timer interrupts are really useful. Previously, we've been using the sleep() command to set the timing of a program. The problem with this is you can only control one thing with this method: the sleep() command locks up the processor while it's executing. In contrast, with timer interrupts, you only call a subroutine when needed - i.e. when a fixed amount of time has gone by.

In this way, you can control the timing of many things at once. With timer interrupts, you can

- Call a subroutine every 10ms to set the sampling rate, (periodic event) or
- Call a subroutine 1 second in the future (one-time event or a one-shot)

Since interrupts don't interfere with other interrupts, multiple interrupts can be set up (for example, you could play two tunes at the same time.)



Two types of timer interrupts: periodic (repeat every T seconds) or one-shot (happen one time T seconds in the future)

In this lecture, we'll go over

- What Timer interrupts are,
- How they're set up,

and some things you can do with timer interrupts, such as

- Implement a Ton() function: a button has to be held down for T seconds to register
- Implement a Toff() function: an output remains on for T seconds after it's turned off
- Set the timing for a stoplight using timer interrupts, and
- Set the of a game precisely: play *hungry hungry hippo* for 10.00 seconds

Turning On Periodic Timer Interrupts

Starting out, let's just turn on a timer interrupt and have it count every 1.00 second. Timer interrupts are similar to edge interrupts:

- The interrupt subroutine is called by a hardware event (one second elapses in this case).
- As such, global variables are needed to transfer data to and from the main routine
- In-between interrupts, the main routine is free to do whatever it wants (you're not stuck in a wait loop waiting for one second to elapse

```
1   from machine import Pin, Timer
2   from time import sleep_ms
3
4   led = Pin(17, Pin.OUT)
5   tim = Timer()
6   N = 0
7
8   def tic(timer):
9       global N
10      N += 1
11
12  tim.init(freq=1, mode=Timer.PERIODIC, callback=tic)
13
14  while(1):
15      print(N)
16      sleep_ms(100)
```

Basic Timer interrupt routine: The interrupt is called every 1.00 second (freq = 1Hz)

Line 12 sets up the interrupt.

```
tim.init(freq=1, mode=Timer.PERIODIC, callback=tick)
```

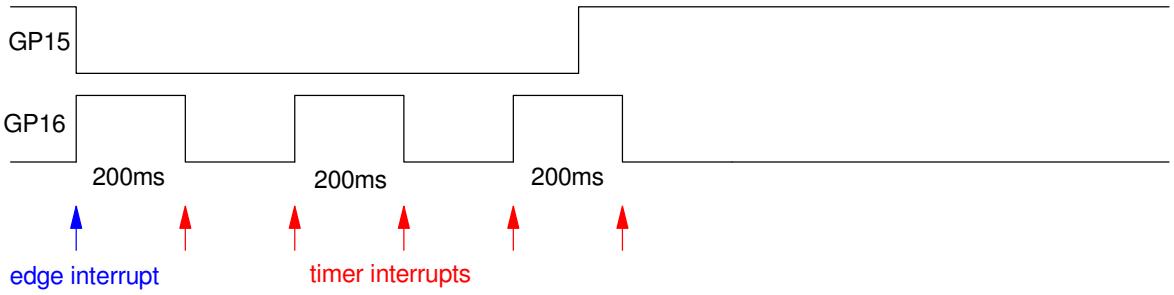
The terms are:

- freq: the frequency of the interrupt in Hz. (or, the time between interrupts is T = 1 / Hz seconds)
- mode=Timer.PERIODIC: the interrupt is called over and over again
- mode=Timer.ONE_SHOT: the interrupt is only called once
- callback=tic: the name of the interrupt service routine is tic

Fire Cheat:

As an interrupt, data cam be saved and shared with other routines using global variables. For example, write a program for a fire-button cheat:

- When GP15 is pressed, (falling edge interrupt)
- Three pulses are output of GP16 (three shots using a timer interrupt)
- Each shot should be on for 200ms and off for 200ms



Fire Control Cheat: Each time you press GP15, three shots are output on GP16

The code includes a counter (global variable N):

- On the edge interrupt, N is initialized to 5 (five timer interrupts are needed)
- Each timer interrupt, N is decremented to zero, stopping at zero
- When N=1, the timer interrupt is changed to being a one-shot event, wrapping up this sequence

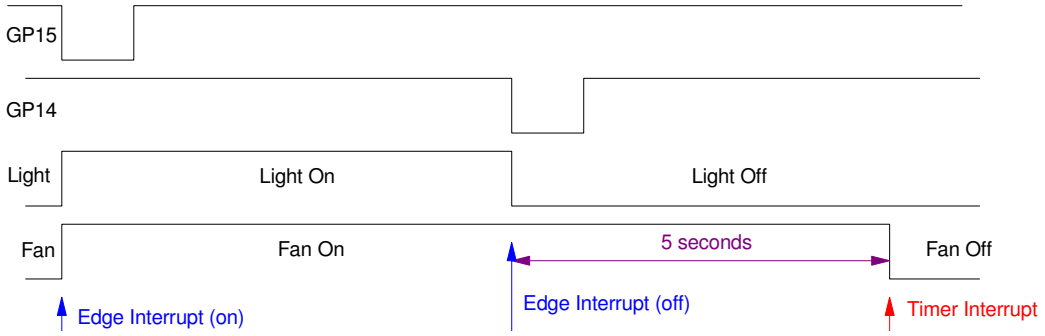
Code:

```
# Fire Control Cheat

from machine import Pin, Timer
from time import sleep_ms

tim = Timer()
N = 0
pin1 = Pin(15,Pin.IN,Pin.PULL_UP)
LED  = Pin(16,Pin.OUT)

def Fire(pin1):
    global N
    LED.value(1)
    N = 5
    tim.init(freq=5,mode=Timer.PERIODIC,callback=Tic)

def Tic(timer):
    global N
    if(N):
        N -= 1
        LED.toggle()
        if(N == 1):
            tim.init(freq=5,mode=Timer.ONE_SHOT,callback=Tic)
    else:
        LED.value(0)

pin1.irq(trigger=Pin.IRQ_FALLING, handler=Fire)

while(1):
    print(LED.value(), N)
    sleep_ms(100)
```

Toff: Bathroom Light & Fan:

The ONE_SHOT feature allows you to set up events for N seconds in the future. For example, write a program for a bathroom light and fan:

- When GP15 is pressed (goes low), both the bathroom light and fan turn on.
- When GP14 is pressed (goes low), the light turns off immediately but the fan remains on for 5 seconds.



Turn on the light and fan when the ON button is pressed. Turn off the fan five seconds after the OFF button is pressed.

Here, three interrupts are used:

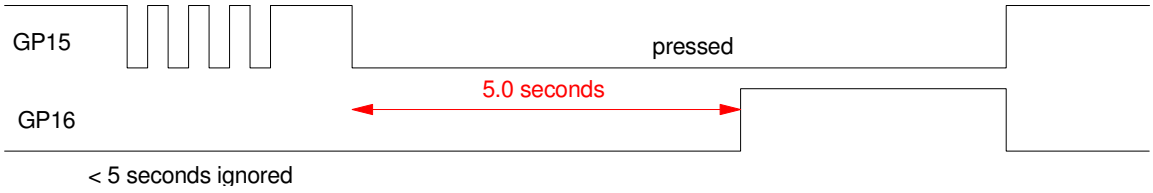
- Falling edge on GP15 turns on the light and fan
- Falling edge on GP14 turns off the light and starts a Timer interrupt that's a one-time event (one_shot), 5 seconds in the future
- Timer interrupt: turns off the fan (five seconds after the GP14 interrupt)

Python Code: There are many ways to program this. With interrupts, it's pretty simple and elegant:

```
from machine import Pin, Timer
from time import sleep_ms

tim = Timer()
N = 0
pin1 = Pin(15,Pin.IN,Pin.PULL_UP)
pin2 = Pin(14,Pin.IN,Pin.PULL_UP)
LED  = Pin(16,Pin.OUT)
Fan  = Pin(17,Pin.OUT)

def On(pin1):
    LED.value(1)
    Fan.value(1)

def Off(pin2):
    LED.value(0)
    tim.init(freq=1/5,mode=Time.ONE_SHOT,callback=Tic)

def Tic(timer):
    Fan.value(0)

pin1.irq(trigger=Pin.IRQ_FALLING, handler=On)
pin2.irq(trigger=Pin.IRQ_FALLING, handler=Off)

while(1):
    print(LED.value(), Fan.value())
    sleep_ms(100)
```

Ton: Turn on if a button is held down for X seconds

In some cases, you want to make sure the operator meant to press the on button. One way to do this is to require that the button is held down for a fixed amount of time, such as 5 seconds. Button presses shorter than this are ignored.



Ton: A button must be held down for X seconds to be recognized

There are several ways to do this. One approach is to set up a timer interrupt every 10ms (100Hz).

- If the button is not pressed, a counter (TimeOn) is cleared.
- If the button is pressed, the counter increments each interrupt (10ms)
- If the counter reaches 500 (5.00 seconds), the LED is turned on

Code:

```
# Ton Function (version 1)

from machine import Pin, Timer
import time

light = Pin(16, Pin.OUT)

tim = Timer()
TimeOn = 0

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

def Tic(timer):
    global TimeOn
    if(pin1.value() == 0):
        TimeOn += 1
    else:
        TimeOn = 0
        light.value(0)

    if(TimeOn >= 500):
        TimeOn = 500
        light.value(1)

def LightOff(pin1):
    light.value(0)

tim.init(freq=100, mode=Timer.PERIODIC, callback=Tic)

while (1):
    print(light.value(), TimeOn)
    time.sleep(0.1)
```

While this code works,

- It's a little inefficient (the timer interrupt is called every 10ms needed or not), and
- The timing is a little off (the light is turned on at the 10ms clock tic rather than being synchronized with the edges).

A better solution uses the one-shot feature to

- Set up the light turning on in 5.00 seconds from the falling edge of GP15
- Clearing GP15 and canceling the interrupt on a rising edge (button released)

Code:

```
# Ton Function (ver 2)

from machine import Pin, Timer
import time

tim = Timer()
pin1 = Pin(15,Pin.IN,Pin.PULL_UP)
LED = Pin(16, Pin.OUT)

def Light(pin1):
    if(pin1.value() == 1):
        LCD.value(0)
        tim.init(freq=5,mode=Timer.ONE_SHOT,callback=LightOff)
    else:
        tim.init(freq=1/5,mode=Timer.ONE_SHOT,callback=LightOn)

def LightOn(timer):
    LED.value(1)

def LightOff(timer):
    LED.value(0)

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

while (1):
    print(light.value(), TimeOn)
    time.sleep(0.1)
```

This code is a little trickier - but

- It's more precise: the timer is started at the time the button is pressed
- It's more efficient: you don't need to be calling interrupts every 10ms. They're only called when needed (edges of the button and 5 seconds after a button press)

Hungry-Hungry-Hippo (version 3):

With timer interrupts, you can set the duration of a game more precisely.

- A one-shot can be triggered 10 seconds after the game starts, or
- A repeating interrupt can be called every 10ms so players know how much time is left in the game.

The following code uses the latter method.

```
# Hungry Hungry Hippo (ver 3)

from machine import Pin, Timer
import time

led = Pin(17, Pin.OUT)
tim = Timer()

N1 = 0
N2 = 0
GameTime = 0

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

def tick(timer):
    global led
    global GameTime
    if(GameTime > 0):
        GameTime -= 1
        led.toggle()
    else:
        led.off()

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)

# set interrupt time to 10ms
tim.init(freq=100, mode=Timer.PERIODIC, callback=tick)

GameTime = 1000

while (GameTime > 0):
    print(GameTime*0.01, N1, N2)
    time.sleep(0.1)
```

Shell

```
time score1 score2
0.47 30 18
0.37 30 18
```

StopLight (ver 3)

With timer interrupts, you can set up the time for each color. Building upon the previous stoplight program, write a program that counts from N=0 to 6 with timing being:

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

Interrupts can be used to set these times several ways:

- A timer interrupt can be called every 100ms and keep track of time. At certain times, the lights change (N increases), or
- Each time the light changes, the next interrupt is set up X seconds in the future

Using the latter method:

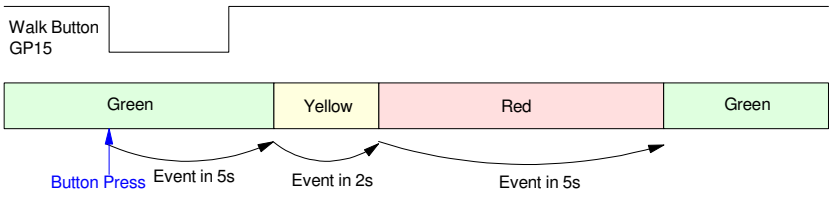
```
# Stoplight (ver 3)

from machine import Pin, Timer
from time import sleep_ms

tim = Timer()
N = 0

def StopLight(pin1):
    global N
    N = (N + 1) % 6
    if(N == 0):
        tim.init(freq=1/5,mode=Timer.ONE_SHOT,callback=StopLight)
    if(N == 1):
        tim.init(freq=1/2,mode=Timer.ONE_SHOT,callback=StopLight)
    if(N == 2):
        tim.init(freq=1/2,mode=Timer.ONE_SHOT,callback=StopLight)
    if(N == 3):
        tim.init(freq=1/5,mode=Timer.ONE_SHOT,callback=StopLight)
    if(N == 4):
        tim.init(freq=1/2,mode=Timer.ONE_SHOT,callback=StopLight)
    if(N == 5):
        tim.init(freq=1/2,mode=Timer.ONE_SHOT,callback=StopLight)

time.init(freq=1/5,mode=Timer.ONE_SHOT,callback=StopLight)

Time = 0
while(1):
   print(Time, N)
   Time += 1
   speep_ms(1000)
```

Shell

```
0    0
1    0
2    0
3    0
4    0
5    1
```

StopLight (ver 4): Adding a Walk button

A simple variation is to have the light always green E/W unless the walk button is pressed. Once that happens, go through the sequence of N=0..5 then stop again at 0 until the walk button is pressed again.



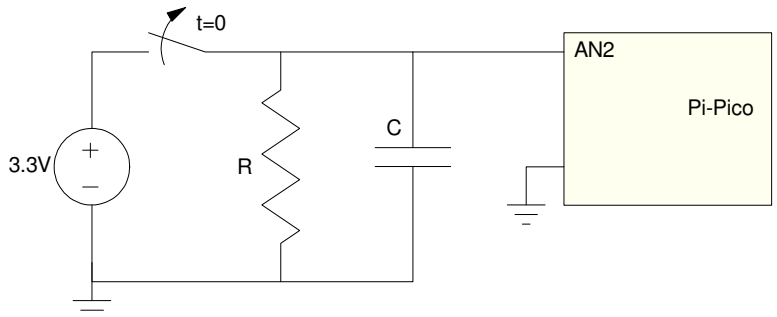
Walk Button: Trigger a light change using one-time events (one-shot interrupts)

```
# Stoplight (ver 3)

from machine import Pin, Timer
from time import sleep_ms

tim = Timer()
N = 0

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

def Walk(pin1):
    tim.init(freq=1/5,mode=Timer.ONE_SHOT,callback=StopLight)

def StopLight(pin1):
    global N
    N = (N + 1) % 6
    if(N == 1):
        tim.init(freq=1/2,mode=Timer.ONE_SHOT,callback=StopLight)
    if(N == 2):
        tim.init(freq=1/2,mode=Timer.ONE_SHOT,callback=StopLight)
    if(N == 3):
        tim.init(freq=1/5,mode=Timer.ONE_SHOT,callback=StopLight)
    if(N == 4):
        tim.init(freq=1/2,mode=Timer.ONE_SHOT,callback=StopLight)
    if(N == 5):
        tim.init(freq=1/2,mode=Timer.ONE_SHOT,callback=StopLight)

pin.irq(trigger=Pin.IRQ_FALLING,handler=Walk)

Time = 0
while(1):
   print(Time, N)
   Time += 1
   speep_ms(1000)
```

The program is now stuck in state N=0 until you press GP15 (the walk button). Once pressed, it goes through the color change sequence and locks up at N=0 until the walk button is pressed again.

Setting a Fixed Sampling Rate with Timer Interrupts

Timer interrupts are a more precise way of setting the sampling rate. For example, let's use the analog input to record the discharge of a capacitor:



Using the analog input to measure the voltage of a capacitor as it discharges

The *time.sleep()* works, but the sampling rate is erratic: the Pi-Pico is doing other things in the background which cause the sampling rate to vary. In addition, the sampling rate isn't exactly 1ms: lines 11..16 take some time to execute, messing up the sampling rate

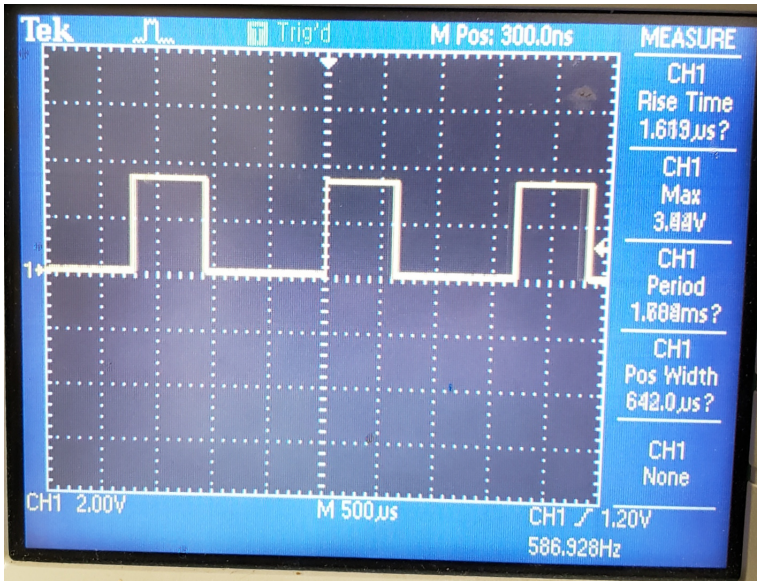
```
1   # Sampling a capacitor discharge (take 1)
2   from machine import ADC
3   from time import sleep
4
5   a2d2 = machine.ADC(2)
6   LED = Pin(16,Pin.OUT)
7
8   k = 3.3 / 65520
9   dT = 0.001
10
11  while(1):
12      LED.value(1)
13      a2 = a2d2.read_u16()
14      V2 = k*a2
15      if((V2 < 3) * (V2 > 0.1)):
16          print('{: 10.3f}'.format(V2))
17      LED.value(0)
        sleep(dT)
```

One way to sample a voltage every 1.00ms. The timing is off due to the execution time of lines 11..16

The timing error shows up on an oscilloscope connected to GP16 (LED in the program).

The period is 1.64ms, meaning

- The main routine takes 640us to execute (when not printing), and
- The sampling rate is off (it should be 1.000ms)

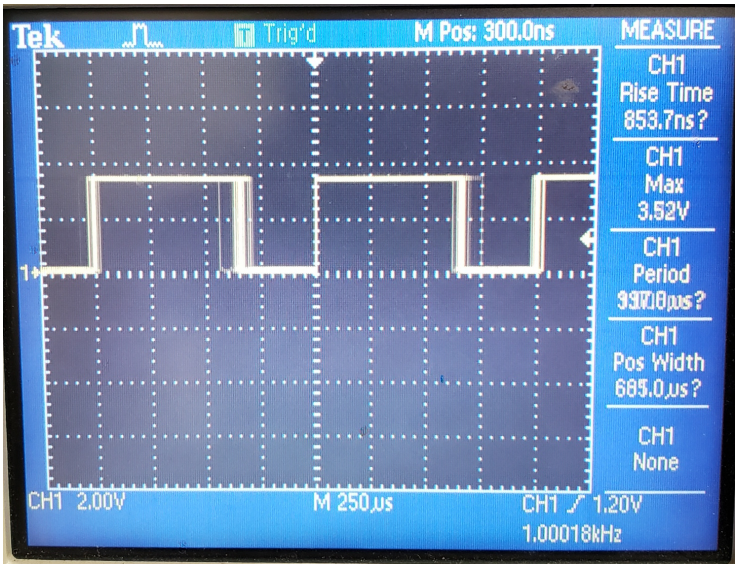


Using a sleep() function to set the sampling rate results in 586Hz sampling (should be 1kHz)

A better way to set the sampling rate is to use a timer interrupt.

- Every 1.00ms, the interrupt is called. This sets a flag every 1.00ms
- The main routine waits until the flag is set. That's the indication it's time to run through the main loop.

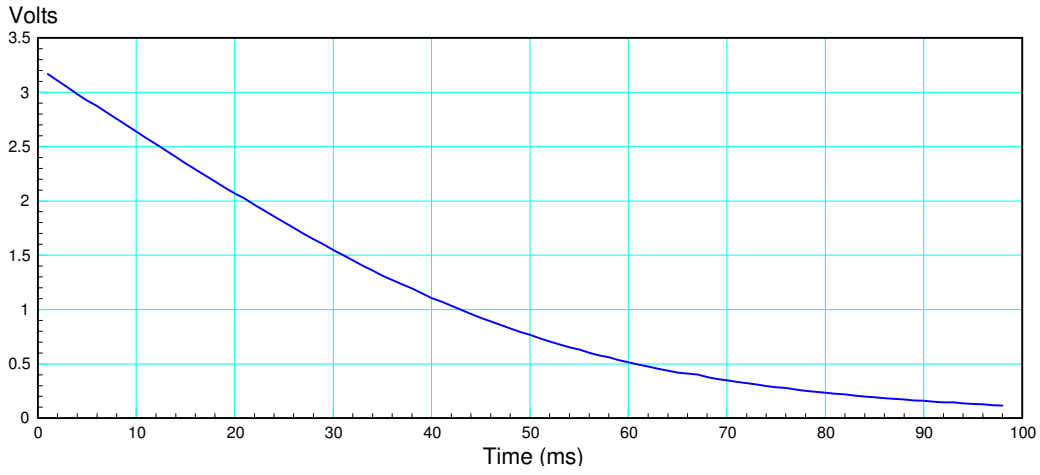
By setting a pin (GP16) just before the main loop starts and clearing it when the main loop ends, you can see how long the main loop takes to execute as well as how much time is spent in the while() loop, waitin g for 1.000ms to elapse. On an oscilloscope, the main loop takes 146us,



Using timer interrupts results in the sampling rate being 1.000kHz as desired
The blurry lines indicate the program execution time varies

```
# Read A/D channel 2 every 1.00ms

from machine import ADC, Pin, Timer
from time import sleep

a2d2 = machine.ADC(2)
tim = Timer()

flag = 0
k = 3.3 / 65520
dT = 0.001

def tick(timer):
    global flag
    flag = 1

tim.init(freq=1000, mode=Timer.PERIODIC, callback=tick)

LED = Pin(16,Pin.OUT)

while(1):
    LED.value(0)
    while(flag == 0):
        pass
    LED.value(1)
    flag = 0

    a2 = a2d2.read_u16()
    V2 = k*a2

    print('{: 7.4f}'.format(V2))
```

Better way to record voltages every 1.00ms



Voltage vs. time as a capacitor discharges: sampling rate set using a timer interrupt

Digital Filters

Yet another application of timer interrupts is implementing digital filters. With a digital filter, you *have* to know the sampling rate, T: the conversion from the s-plane to the z-plane depends upon knowing T. If T changes, the filter is wrong.

For example, design a digital filter to implement

$$Y = \left(\frac{50}{(s+5+j5)(s+5-j5)} \right) X = \left(\frac{50}{s^2+10s+50} \right) X$$

One way to do this is to convert to the z-plane

$$z = e^{sT}$$

Assuming a sampling rate of 10ms (T = 0.01), the poles in the s-plane convert to the z-plane as

$$s = -5 + j5 \qquad z = e^{sT} = 0.9500 + j0.0475$$

$$s = -5 - j5 \qquad z = e^{sT} = 0.9500 - j0.0475$$

so a discrete-version of G(s) would be

$$Y = \left(\frac{k(z+1)^2}{(z-0.9500-j0.0475)(z-0.9500+j0.0475)} \right) X$$

where k is chosen to set the DC gain to 1.00 (same as G(s)). Multiplying out

$$Y = \left(\frac{0.001209\left(z^2+2z+1\right)}{z^2-1.900z+0.904837} \right) X$$

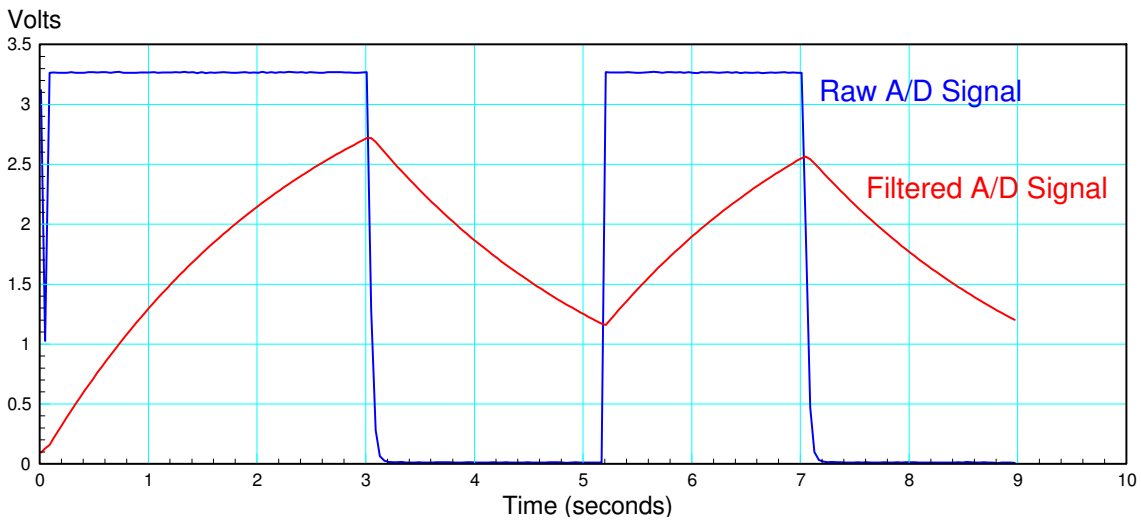
In code

```
x2 = x1
x1 = x0
x0 = a/d reading

y2 = y1
y1 = y0
y0 = 1.9008*y1 - 0.90483*y2 + 0.001209*(x0 + 2*x1 + x2)
```

or with everything else...

```
from machine import ADC, Pin, Timer

a2d2 = machine.ADC(2)
tim = Timer()

flag = 0
k = 3.3 / 65520
x0 = x1 = x2 = y0 = y1 = y2 = 0

def tick(timer):
    global flag
    flag = 1

tim.init(freq=100, mode=Timer.PERIODIC, callback=tick)

LED = Pin(16,Pin.OUT)


while(1):
    LCD.value(0)
    while(flag == 0):
        pass
    LED1.value(1)
    flag = 0

    a2 = a2d2.read_u16()
    V2 = k*a2

    x2 = x1
    x1 = x0
    x0 = k*a2

    y2 = y1
    y1 = y0
    y0 = 1.9008*y1 - 0.90483*y0 + 0.001209*(x0 + 2*x1 + x2);

    print('{: 7.4f}'.format(x0), '{: 7.4f}'.format(y0))
```

Code for implimenting a low-pass filter with poles at s = -5 + j5



Result of filtering the A/D signal

Stepper Motor Controls

Finally, timer interrupts are also useful for controlling the position of a stepper motor. The following code uses a timer interrupt to step a motor. Every 10ms the interrupt is triggered and

- Steps forward one step if the actual angle (Step) is less than the desired angle (Ref),
- Steps backward one step if Step > Ref, and
- Does nothing if Step is equal to Ref

By placing this inside the interrupt, the main routine is free to do whatever you want - in this case monitoring the push buttons. (note that monitoring the push buttons could also be moved to an interrupt, leaving nothing for the main routine to do.

```
# Stepper Motor Control

from machine import Pin, Timer
import time

tim = Timer()
Step = 0
Ref = 0

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

Sa = Pin(6,Pin.OUT)
Sb = Pin(7,Pin.OUT)
Sc = Pin(8,Pin.OUT)
Sd = Pin(9,Pin.OUT)

def Stepper(Step):
    X = Step % 4;
    if(X == 0):
        Sa.value(1)
        Sb.value(0)
        Sc.value(0)
        Sd.value(0)
    if(X == 1):
        Sa.value(0)
        Sb.value(1)
        Sc.value(0)
        Sd.value(0)
    if(X == 2):
        Sa.value(0)
        Sb.value(0)
        Sc.value(1)
        Sd.value(0)
    if(X == 3):
        Sa.value(0)
        Sb.value(0)
        Sc.value(0)
        Sd.value(1)

def tick(timer):
    global Step, Ref
    if(Step < Ref):
        Step += 1
    if(Step > Ref):
        Step -= 1
    Stepper(Step)

tim.init(freq=100, mode=Timer.PERIODIC, callback=tick)

while (1):
    if(pin1.value() == 0):
        Ref = 0
    if(pin2.value() == 0):
        Ref = 100
    print(Ref, Step, )
    time.sleep(0.1)
```