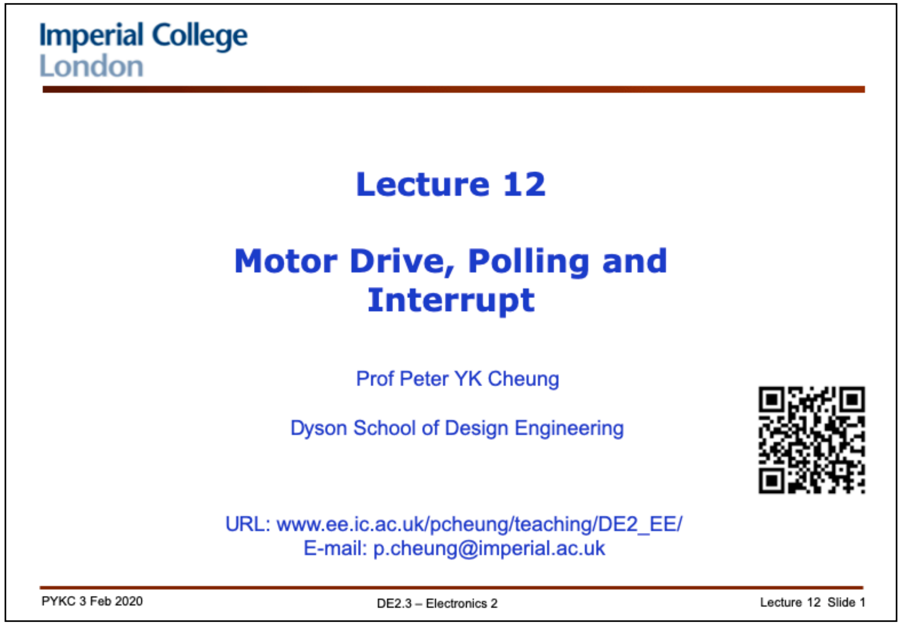
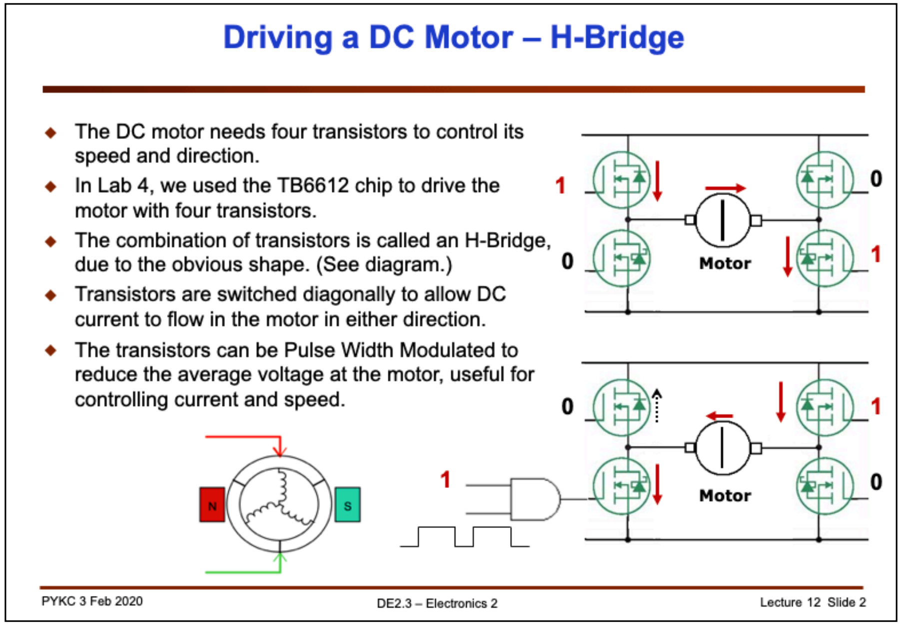


In this lecture, we will go through what you have learned in Lab 4. In particular, I will explain the limitation of polling as a method of detecting a real-time event. Then I will explain what and why interrupt is so much better.

Finally I will explain how you can use interrupt with the Pybench board using Micropython.



The content of this slide was presented to you last year in DE1.3 Electronics 1 module in Lecture 14 – Drive. I present this here again to refresh your memory.

Since motor coils are essentially inductors, they have low DC impedances (resistance of the wiring). Hence when driving motors, we need to use special driver chips.

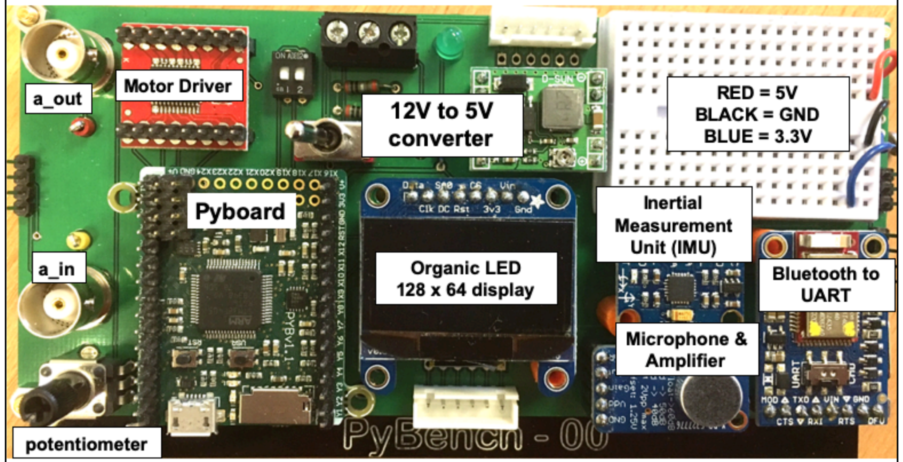
The driver chip you will use in Lab 4 (the TB6612) is often called the H-Bridge Driver. Shown here is the simplified block diagram. There are four transistors connected to the supply rail and ground. (It doesn't matter which is which because the circuit is symmetrical.) The motor is connected in the middle forming the horizontal link of the H. The transistors are MOSFETs (metal oxide silicon field effect transistors) which is acting like a voltage controlled switch. When a '1' or high voltage is applied to the gate control terminal, the transistor turns ON and conduct electricity. If a '0' or low voltage is applied, the transistor is OFF. So the top diagram shows a configuration that results in the supply voltage being applied to the left terminal of the motor. The right terminal of the motor is grounded, and the motor turns in one direction. Reversing the control to the transistors results in the motor turning in the other direction.

If you use an AND gate at the control input, you can also add a PWM signal to control the speed of the motor.

Basically the '1' and '0' control signals are the A0 and A1 signals on the TB6612. The PWM signal is what you apply to the input of the AND gate.

Now you know how the TB6612 works.

Pybench Board and its components

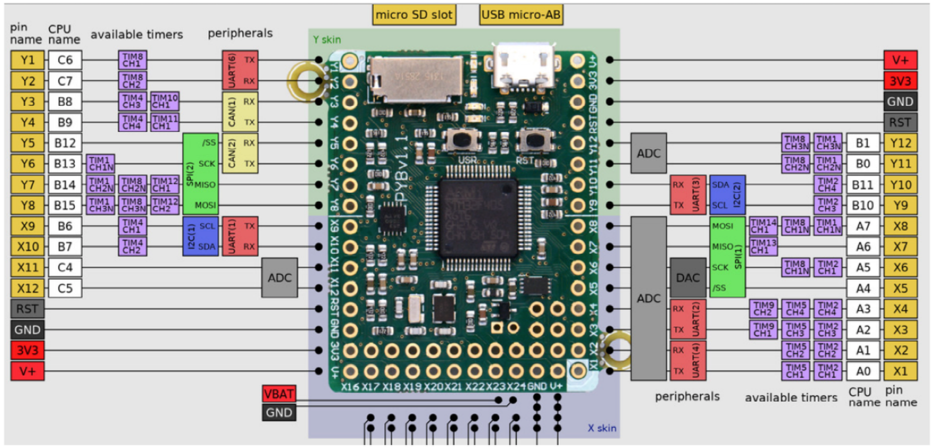


Last year, you wired your motors to the TB6612 H-bridge driver chip (the one in brown colour) yourself. This year, the wirings are all done for you on the Pybench PCB.

You will also be connecting the Pybench board to a 12V lithium battery via the large yellow power connectors. The toggle switch turns this ON and OFF.

The motor is connected to the Pybench board via the 6-way ribbon cable. You will be learning how to drive the motors (again) in Lab 4 on Wednesday.

You will also be using the Bluetooth board (BluFruit UART from Adafruit) to control the speed of the motor. Here are the pin names on the Pyboard for reference.



Driving the motor with TB6612

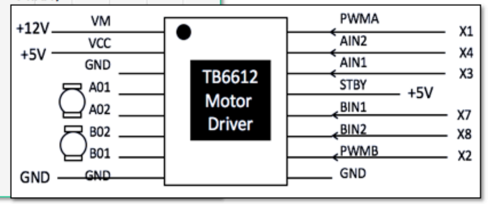
```python
import pyb
from pyb import Pin, Timer

# Define pins to control motor
A1 = Pin('X3', Pin.OUT_PP)      # Control direction of motor A
A2 = Pin('X4', Pin.OUT_PP)
PWMA = Pin('X1')                # Control speed of motor A

# Configure timer 2 to produce 1KHz clock for PWM control
tim = Timer(2, freq = 1000)
motorA = tim.channel (1, Timer.PWM, pin = PWMA)

def A_forward(value):
    A1.low()
    A2.high()
    motorA.pulse_width_percent(value)

A_forward(50)
```



Exercise 1 of Lab 4 is just a revision from last year’s Electronic 1 module. If you have forgotten this, please go back to last year’s lecture on “Drive”.

Here are some interesting questions to ask yourself to check whether you have learned what is expected of you:

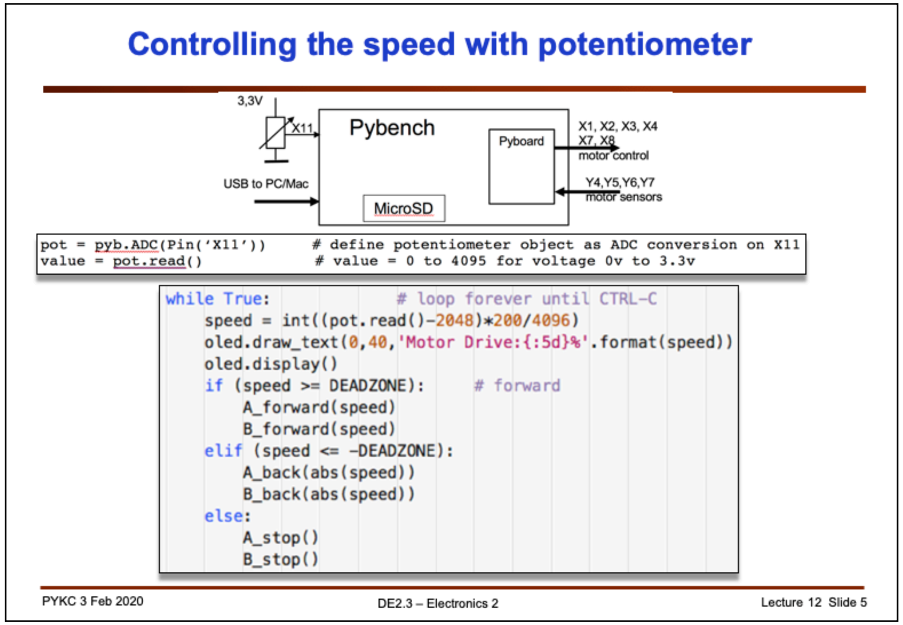
1. Why do you need this driver chip at all? Could you drive the motor directly from the microprocessor?
2. How are the two pins (IN1 and IN2) used to control the direction of the motor?
3. What is PWM and why is it desirable to use PWM to control the speed of the motor instead of using analogue voltage level (e.g. from a DAC signal)?
4. What is meant by “Creating a pin object A1” in the Python code?

```python
A1 = Pin('X3', Pin.OUT_PP)
```

5. Explain how timer 2 is programmed to produce the PWM signal to drive motor in the following lines.

```python
# Configure timer 2 to produce 1KHz clock for PWM control
tim = Timer(2, freq = 1000)
motorA = tim.channel (1, Timer.PWM, pin = PWMA)
```

6. How should you choose the frequency of the PWM signal to drive the motor?



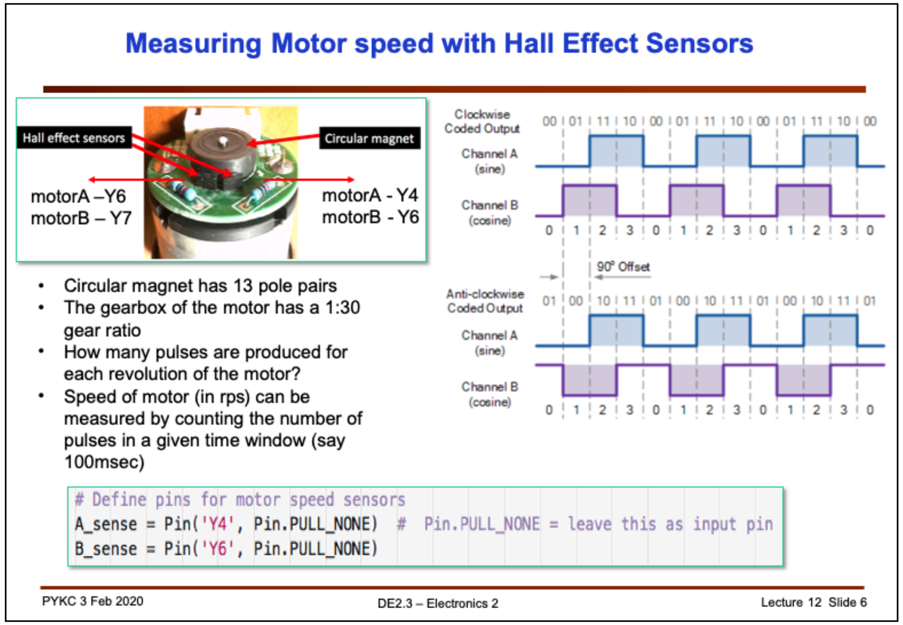
Next, we use the potentiometer (5kΩ) to control motor speed and direction. Here are the questions to test yourself:

1. In Micropython, how do you create an object to perform ADC conversion? Why in this case, we use pin X11?
2. How do you define and work out the resolution of the ADC converter?
3. Explain the meaning of the statement:

```
speed = int((pot.read()-2048)*200/4096)
```

4. Explain the meaning of the format statement in Python:

```
oled.draw_text(0,40,'Motor Drive:{:5d}%'.format(speed))
```



Next, we use the Hall Effect Sensors (two) on the motor to determine the speed of the motor and direction of the motor. The questions to ask yourself are:

1. Refer to the sensor output signals, what happens when you increase the motor speed?
2. How would the two sensor signals differ when you change the direction of rotation in the motor?
3. Given the waveform of the two signals (Channel A and B) from the sensors, the relative phase is always $\pm\pi/2$. Why?
4. Given the circular magnet has 13 pole pairs, and that the gear of the motor has a 1:30 reduction ratio, how can you derive the speed of motor (in revolutions per second) from the number of rising edges E in a period T? (answer: 390 pulses per revolution. Therefore speed of motor is:

   motor_speed (in rps) = (number of pulses/390) / T in seconds

Pseudo code to measure speed by polling

- Initialize variables to zero: motor_speed, sensor_state, pulse_count
- Repeat forever:

```
Mark current time (as tic)
If sensor has gone from low to high (rising edge)
        increment pulse_count
Update sensor_state by reading hall effort sensor value
If elapse_time >= 100ms
        motor_speed = pulse_count
        reset pulse_count
        display speed on OLED as motor_speed/39
```

Discuss: what is the limitation of polling?

This is typically how one can measure the motor speed by polling – continuously checking in a tight loop whether something has happened or not.

Here there are TWO polling operation happening. The first if-statement checks to see if the Hall Effort signal has a rising edge (goes from low to hight). The second if-statement checks for a time window of 100msec. By counting the number of pulses detected in 100ms window, we can calculate the speed of the motor using the formula:

motor_speed (in rps) = number of pulses/39

39 because each revolution of the motor generates 390 pulses. Therefore in a 100msec period, one revolution will give us 39 pulses!

Measure motor speed by polling

- Polling means checking for some event in a loop, then do something
- Here we check sensor signal of motor A changing from low to high in the polling loop
- When this occurs, increment a counter A_count
- We also check elapsed time = 100msec in polling loop (tic-toc)
- If time out, save count as speed measurement A_speed, and reset counter

```python
# Initialise variables
A_state = 0          # previous state of A sensor
A_speed = 0          # latest speed of motor A
A_count = 0          # positive transition count
tic = pyb.millis(); # keep time in millisecond

while True:                # loop forever until CTRL-C
    # detect rising edge on sensor A
    if (A_state == 0) and (A_sense.value()==1): # rising edge detected on A
        A_count += 1
    A_state = A_sense.value()   # read value on pin A_sense

    # Check to see if 100 msec has elapsed
    toc = pyb.millis()
    if ((toc-tic) >= 100):
        A_speed = A_count

        # drive motor - controlled by potentiometer (as before)
        .........

        A_count = 0          # reset transition count

    # Display new speed
        oled.draw_text(0,20,'Motor A:{:5.2f} rps'.format(A_speed/39))
        oled.display()
        tic = pyb.millis()
```


We measure the speed of rotation by counting the number of low-to-high transitions on one of the two Hall Effect Sensor signals.

This can be achieved by polling – checking in the code when such transition has occurred. If yes, up a counter value. Then check if 100msec has elapsed. If yes, remember the count value and reset the counter.

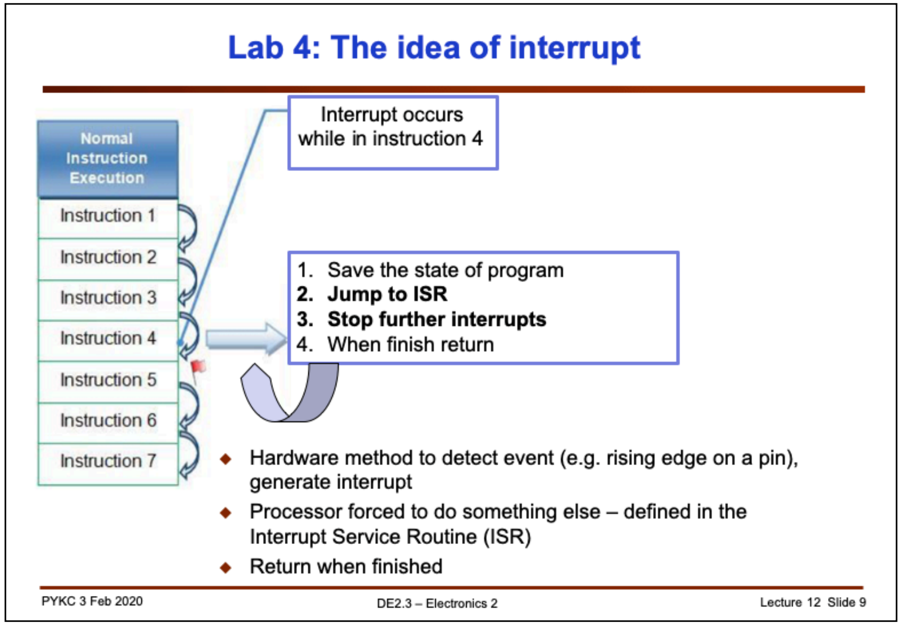
Questions to ask yourself:

1. What is the purpose of these two lines?

```python
if (A_state == 0) and (A_sense.value()==1): # rising edge detect
    A_count += 1
A_state = A_sense.value()   # read value on pin A_sense
```

2. How are tic and toc, which are built-in functions in Matlab, implemented in Micropython?
3. Explain the following codes:

```python
# Check to see if 100 msec has elapsed
toc = pyb.millis()
if ((toc-tic) >= 100):
    A_speed = A_count
```



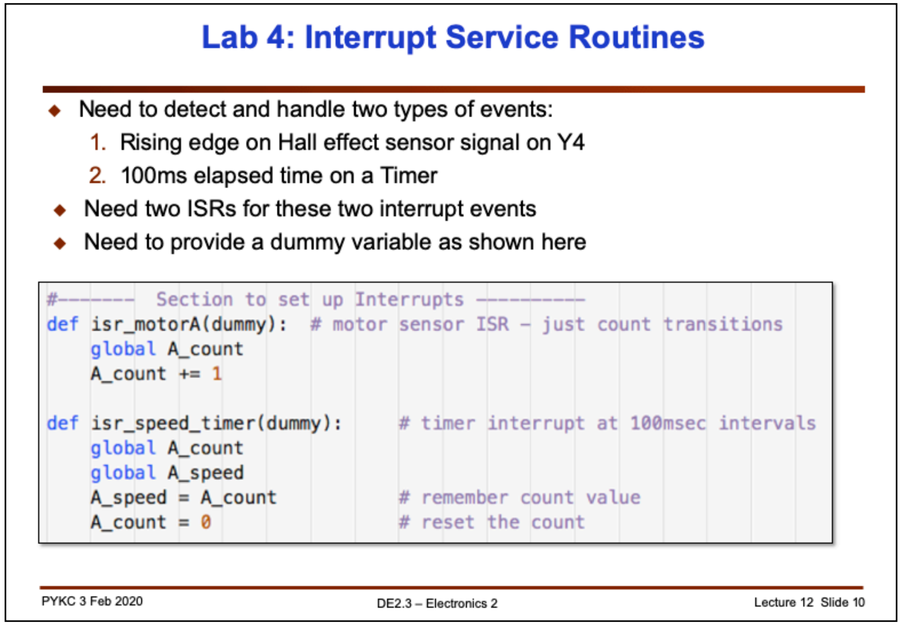
The reason why polling is not a good method to measure speed of motor is that microprocessor can only execute ONE instruction stream at a time. If you are checking (polling) for rising edge, you cannot do other things. Conversely if you are doing other things, you will miss the rising edges. That's why in the experiment, you found that the polling method give a speed reading that is "noisy", meaning that it is jumps all over the place!

Interrupt is different. You use HARDWARE method to detect the occurrence of an event. Forces the processor to suspend whatever it is doing at the time, and go to another segment of CODE to service the interrupt (hence we call this the Interrupt Service Routine or ISR).

When finished, return to the interrupted code and continue as before.

Question to ask yourself:

1. Why is interrupt better than polling?
2. What happens if your interrupt service routine is long and complex?
3. How should you think about a system with multiple interrupts?
4. What is it meant by "saving the state fo the program"? Why is this necessary?



Here are two interrupt service routines. The first to handle low-to-high transition on the senor signal from Motor A. The second to handle timer alarm which happens every 100msec.

Question to ask yourself:

1. When will the functions isr_MotorA and isr_speed_timer be executed?
2. What are the purposes of these two functions?
3. Why you need to define A_count and A_speed as global?

Lab 4: setting up the interrupts

- Allocate some buffer space to handle errors
- Specify Pin Y4 as source of interrupt, rising edge
- Define timer 4 as a 100msec period timer (10Hz)
- timer.callback (ISR) - tell timer to generate an interrupt at end of period, and execute ISR

Specify ISR for timer time-out

Specify ISR for pin rising edge

```
# Create external interrupts for motorA Hall Effect Senor
import micropython
micropython.alloc_emergency_exception_buf(100)
from pyb import ExtInt

motorA_int = ExtInt ('Y4', ExtInt.IRQ_RISING, Pin.PULL_NONE,isr_motorA)

# Create timer interrupts at 100 msec intervals
speed_timer = pyb.Timer(4, freq=10)
speed_timer.callback(isr_speed_timer)
```


How does one set up interrupts in MicroPython using the Pyboard and the Pybench System? First you need to include the following statement to allocate memory to store the state of the program:

```
micropython.alloc_emergency_exception_buf(100)
```

Then you have to tell that hardware that pin Y4 will generate an interrupt on every rising edge, and that the interrupt service routine is isr_motorA:

```
motorA_int = ExtInt ('Y4', ExtInt.IRQ_RISING, Pin.PULL_NONE,isr_motorA)
```

Then, you need to program Timer 4 to time out every 100msec:

```
speed_timer = pyb.Timer(4, freq=10)
```

Finally, you need to tell this Timer that it should generate an interrupt when time out, and run isr_speed_timer:

```
speed_timer.callback(isr_speed_timer)
```

Lab 4 – Interrupt MAGIC

```
while True:                # loop forever until CTRL-C

    # drive motor - controlled by potentiometer
    speed = int((pot.read()-2048)*200/4096)
    if (speed >= DEADZONE):      # forward
        A_forward(speed)
        B_forward(speed)
    elif (speed <= -DEADZONE):
        A_back(abs(speed))
        B_back(abs(speed))
    else:
        A_stop()
        B_stop()

    # Display new speed
    oled.draw_text(0,20,'Motor A:{:5.2f} rps'.format(A_speed/39))
    oled.display()
```

Wheel rotating at 1 rps will produce 39 rising edges in 0.1 sec

- Program loop does not deal with motor sensor edge, not 100msec elapse time
- A_speed will always contain instantaneous speed count

Once interrupt is set up properly, the main program loop only controls the motor. Measuring the speed of motor is done automatically.

The global variable A_speed will contain the correct number of transitions in a 100msec window ALL THE TIME, and updated every 100msec automatically.

Pin Assignments for Pybench

PIN	FUNCTION
X1	Motor PWM_A/Servo 1
X2	Motor PWM_B/Servo 2
X3	Motor control AIN1/Servo 3
X4	Motor control AIN2/Servo 4
X5	Analogue OUTPUT
X6	SW0
X7	Motor control BIN1
X8	Motor control BIN2
X9	IMU-I2C SCL
X10	IMU-I2C SDA
X11	POT5K
X12	Analogue INPUT

PIN	FUNCTION
Y1	BLE-UART Tx
Y2	BLE-UAR Rx
Y3	SW1
Y4	Motor sensor A_A
Y5	Motor sensor A_B
Y6	Motor sensor B_A
Y7	Motor sensor B_B
Y8	OLED-i2C RST
Y9	OLED-I2C SCL
Y10	OLED-I2C SDA
Y11	Microphone amplifier
Y12	Unused

For your information and future reference, here are the pin assignment found on the Pybench board.