Imperial College London

Lecture 10

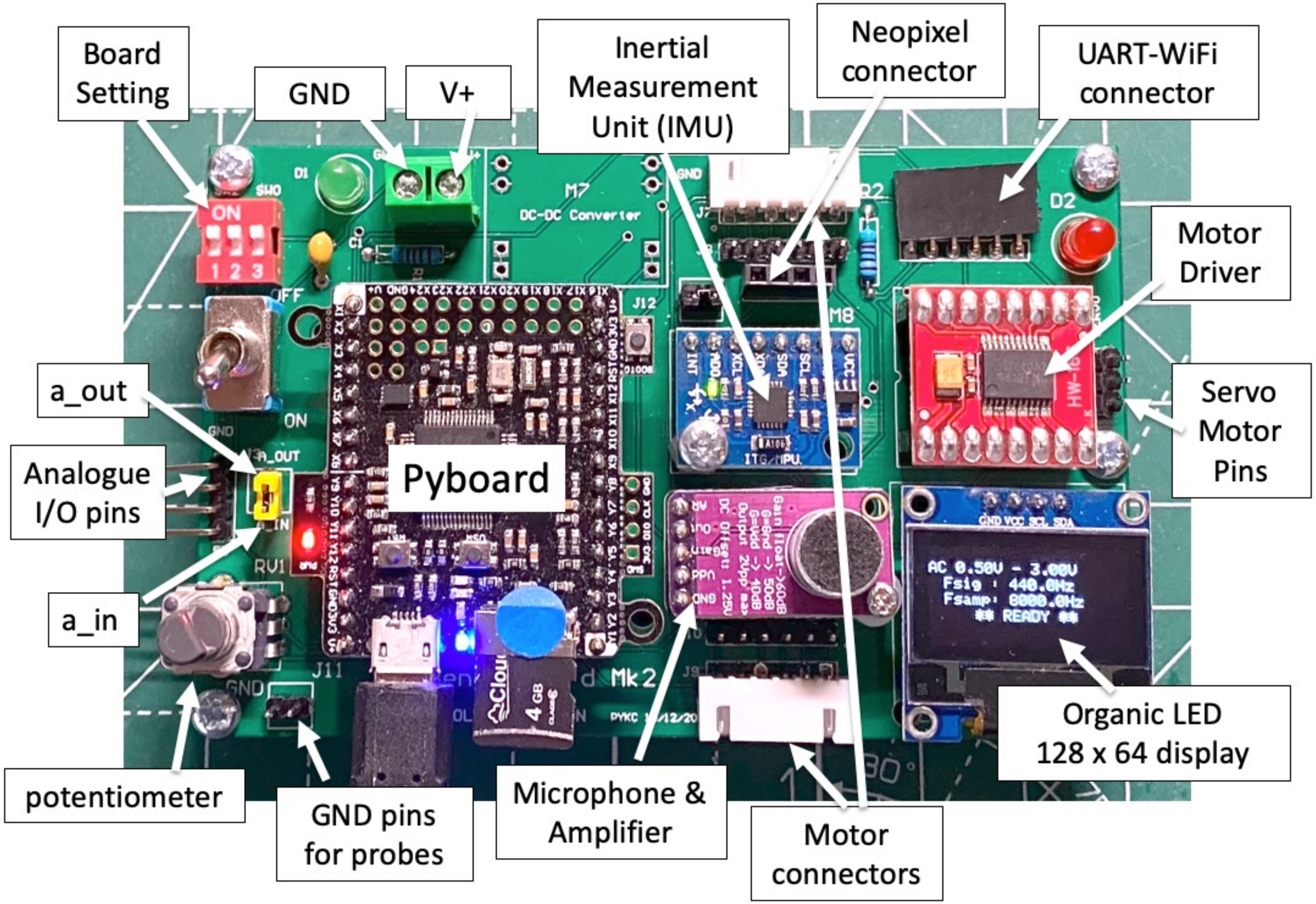
Motor Drive, Polling and Interrupt

Prof Peter YK Cheung

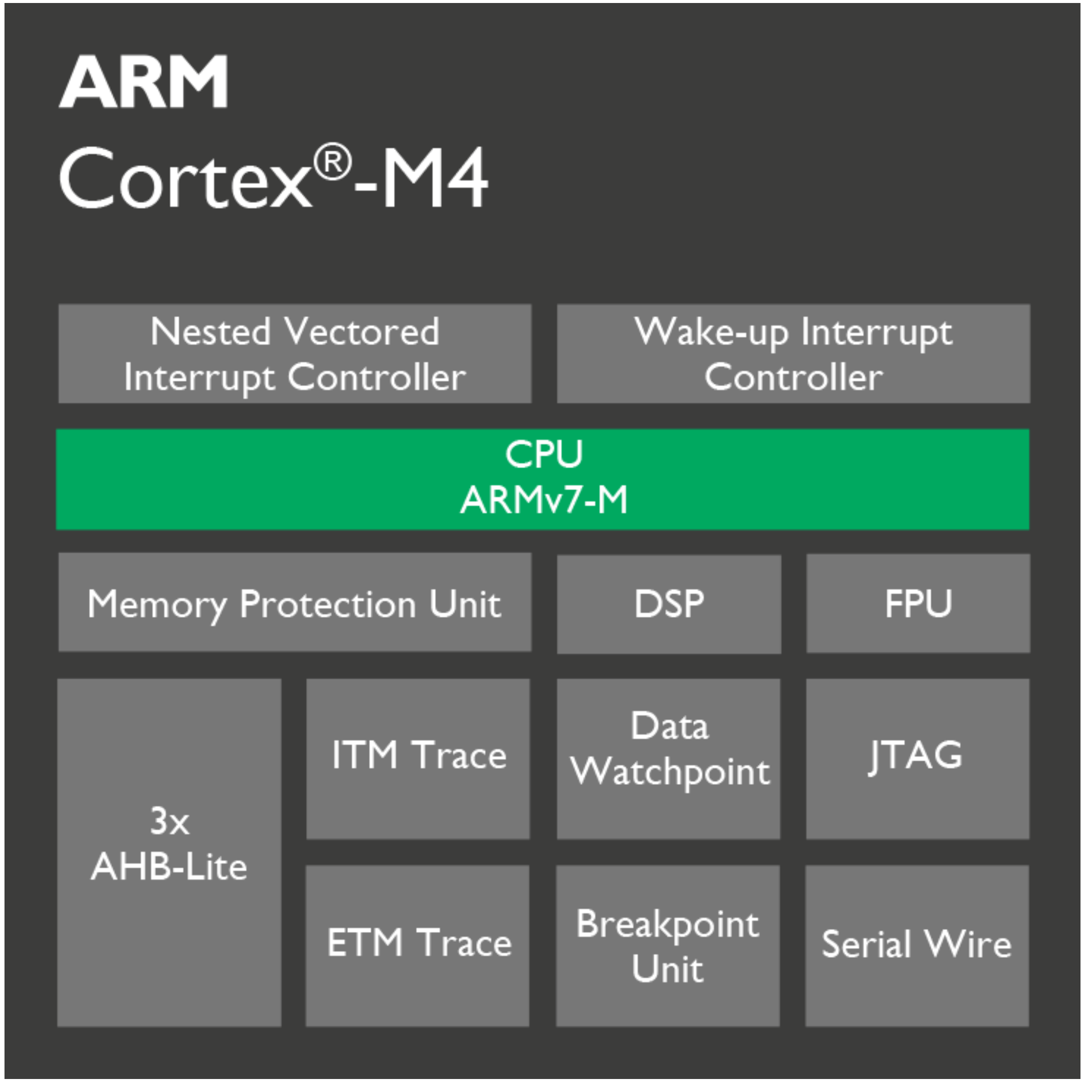
Dyson School of Design Engineering

URL: www.ee.ic.ac.uk/pcheung/teaching/DE2_EE/
E-mail: p.cheung@imperial.ac.uk

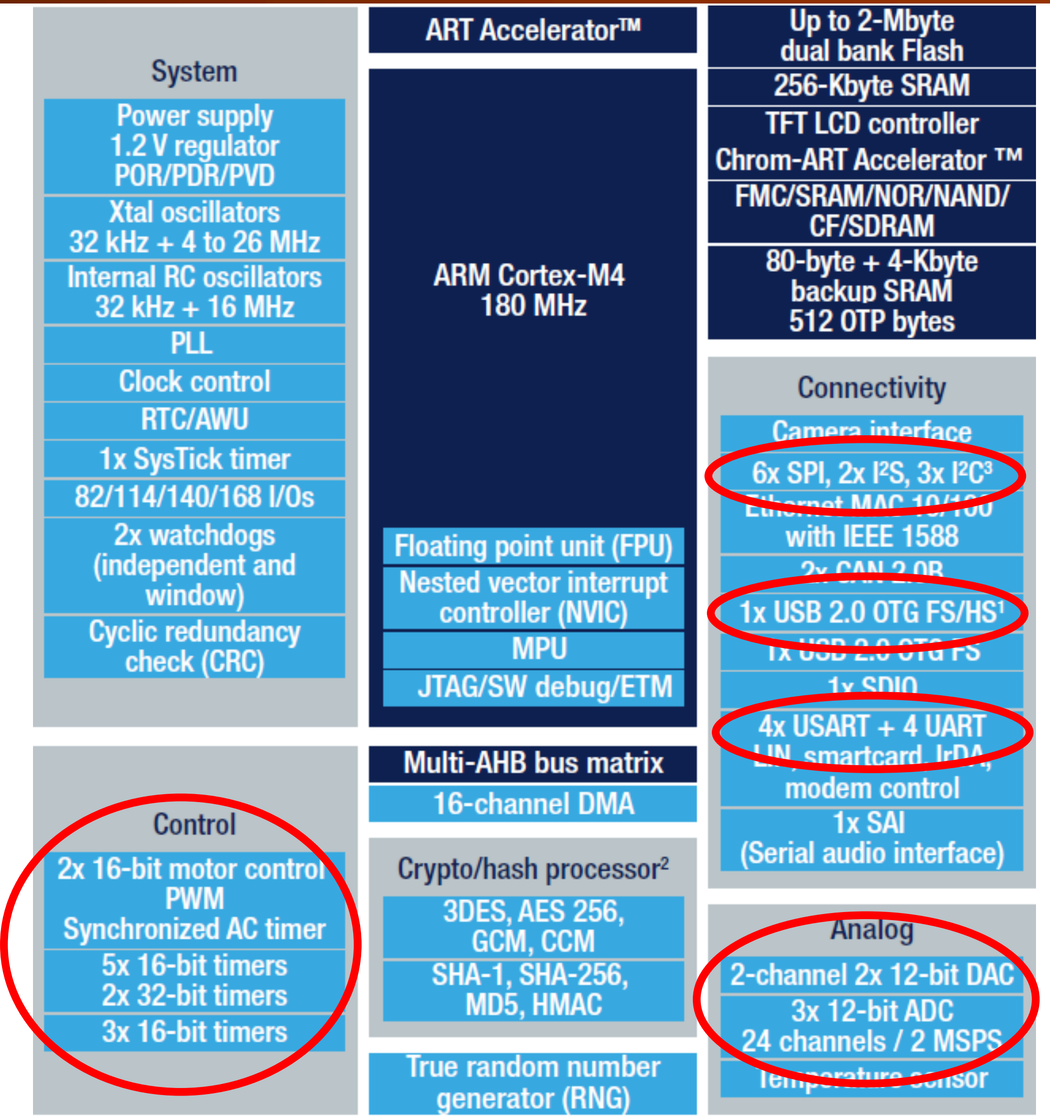
Pybench Board and its components



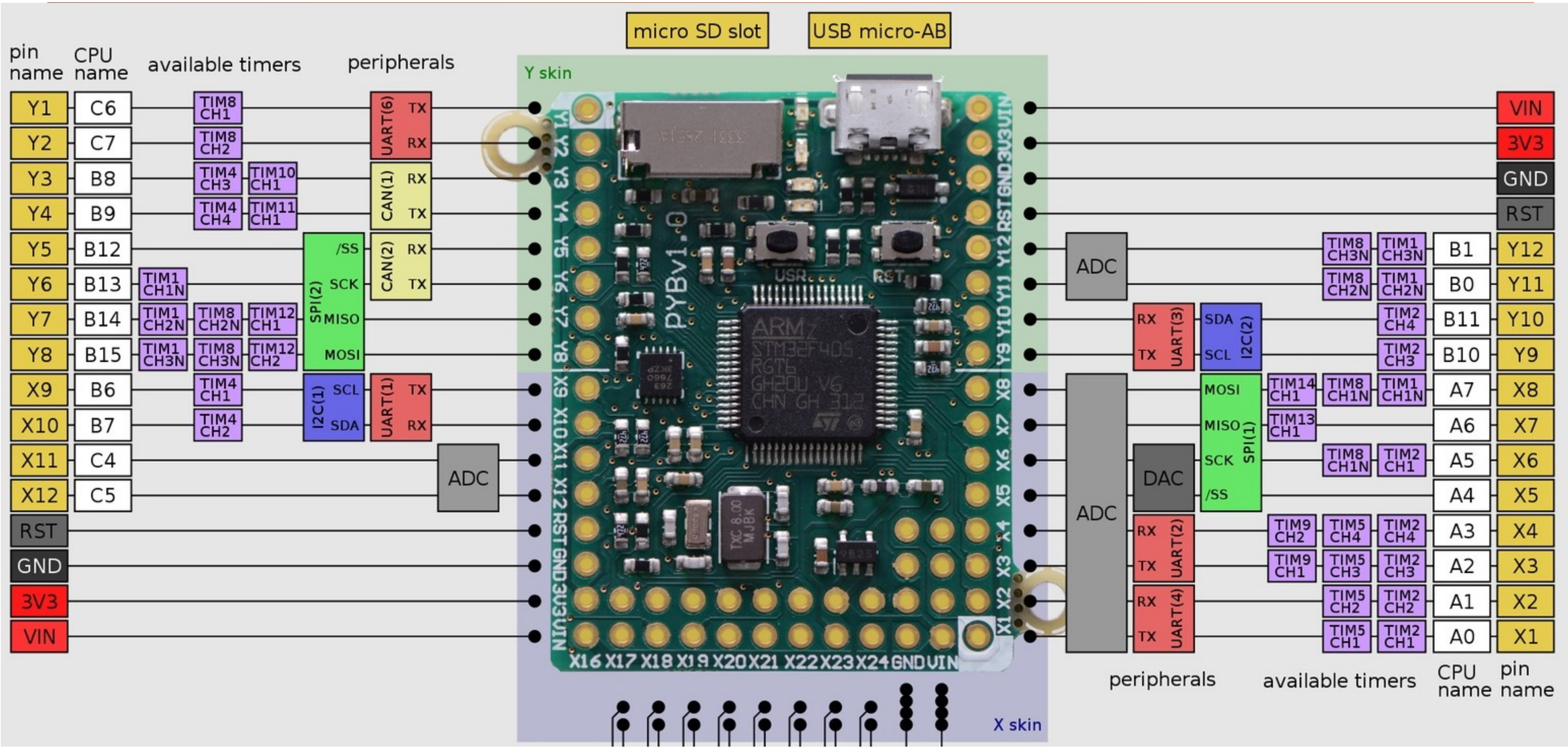
ARM Cortex-M4 Processor



STM32F405 Microcontroller in Pyboard

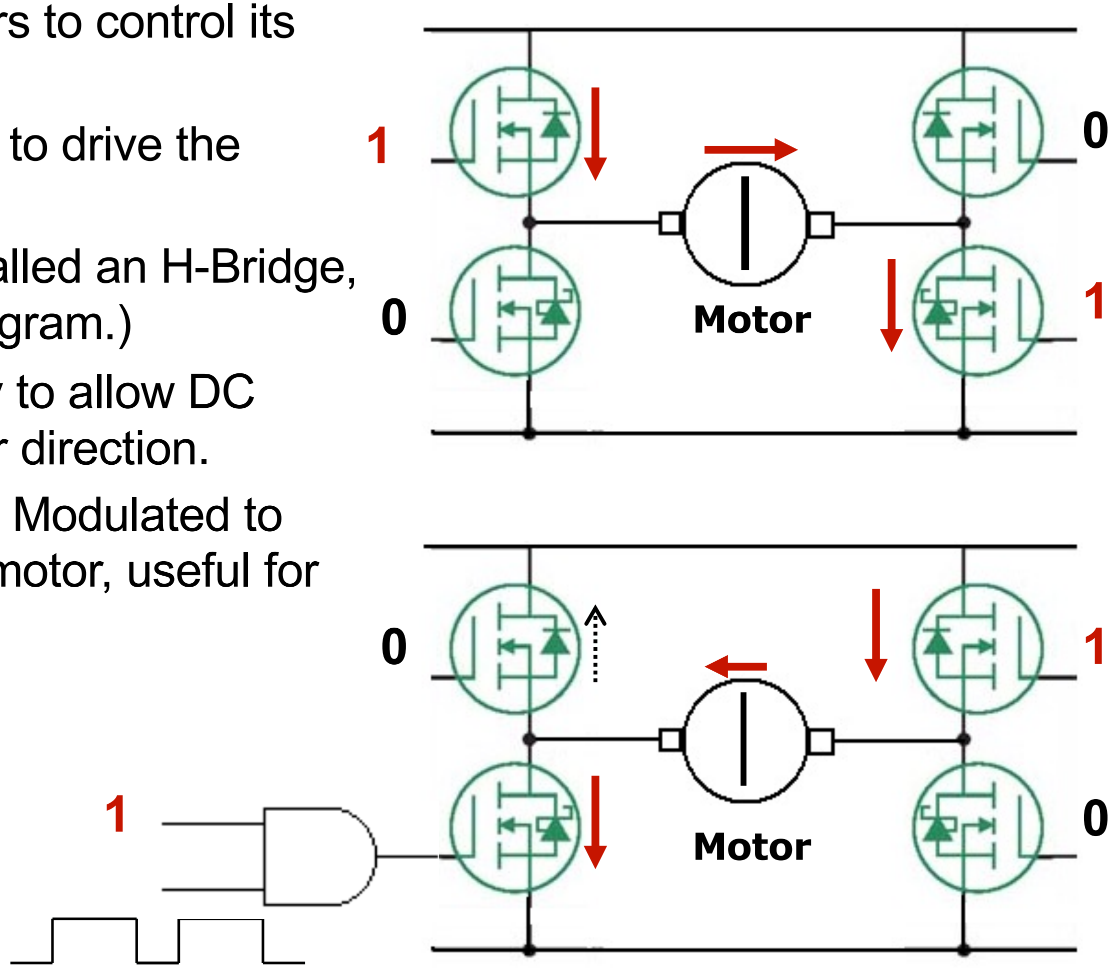


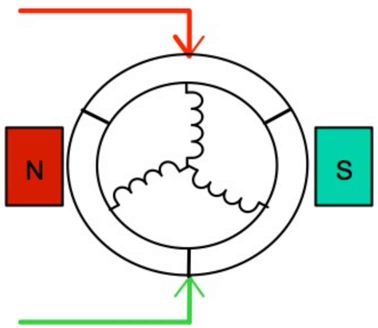
The Pyboard


micro SD slot
USB micro-AB
pin name
CPU name
available timers
peripherals
Y skin
X skin
Y1 C6 TIM8 CH1 UART(6) TX
Y2 C7 TIM8 CH2 UART(6) RX
Y3 B8 TIM4 CH3 TIM10 CH1 CAN(1) RX
Y4 B9 TIM4 CH4 TIM11 CH1 CAN(1) TX
Y5 B12 SPI(2) /SS CAN(2) RX
Y6 B13 TIM1 CH1N SPI(2) SCK CAN(2) TX
Y7 B14 TIM1 CH2N TIM8 CH2N TIM12 CH1 SPI(2) MISO
Y8 B15 TIM1 CH3N TIM8 CH3N TIM12 CH2 SPI(2) MOSI
X9 B6 TIM4 CH1 I2C(1) SCL UART(1) TX
X10 B7 TIM4 CH2 I2C(1) SDA UART(1) RX
X11 C4 ADC
X12 C5 ADC
RST
GND
3V3
VIN
VIN
3V3
GND
RST
Y12 B1 TIM8 CH3N TIM1 CH3N ADC
Y11 B0 TIM8 CH2N TIM1 CH2N ADC
Y10 B11 TIM2 CH4 I2C(2) SDA UART(3) RX
Y9 B10 TIM2 CH3 I2C(2) SCL UART(3) TX
X8 A7 TIM14 CH1 TIM8 CH1N TIM1 CH1N SPI(1) MOSI ADC
X7 A6 TIM13 CH1 SPI(1) MISO ADC
X6 A5 TIM8 CH1N TIM2 CH1 SPI(1) SCK DAC ADC
X5 A4 SPI(1) /SS DAC ADC
X4 A3 TIM9 CH2 TIM5 CH4 TIM2 CH4 UART(2) RX ADC
X3 A2 TIM9 CH1 TIM5 CH3 TIM2 CH3 UART(2) TX ADC
X2 A1 TIM5 CH2 TIM2 CH2 UART(4) RX ADC
X1 A0 TIM5 CH1 TIM2 CH1 UART(4) TX ADC
X16 X17 X18 X19 X20 X21 X22 X23 X24 GND VIN
PYBv1.0
USR
RST

Driving a DC Motor – H-Bridge

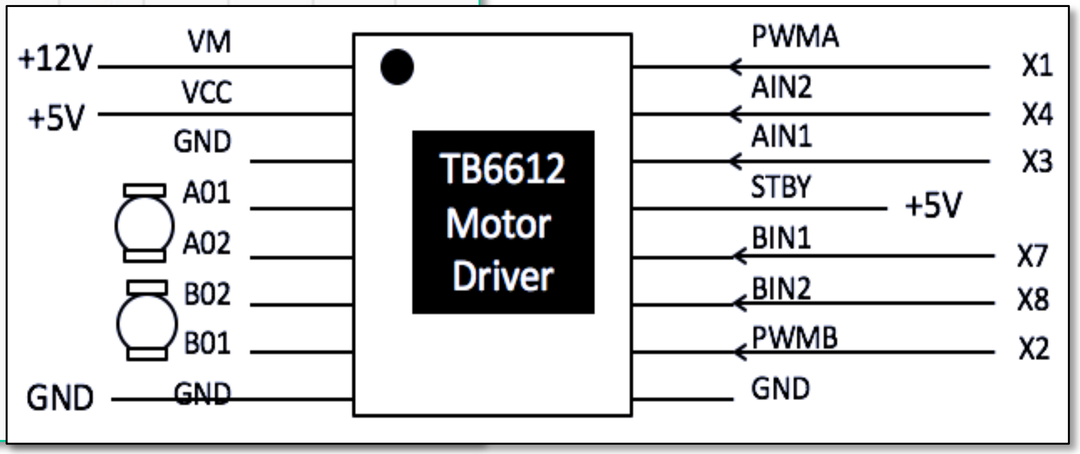
- The DC motor needs four transistors to control its speed and direction.
- In Lab 5, we used the TB6612 chip to drive the motor with four transistors.
- The combination of transistors is called an H-Bridge, due to the obvious shape. (See diagram.)
- Transistors are switched diagonally to allow DC current to flow in the motor in either direction.
- The transistors can be Pulse Width Modulated to reduce the average voltage at the motor, useful for controlling current and speed.



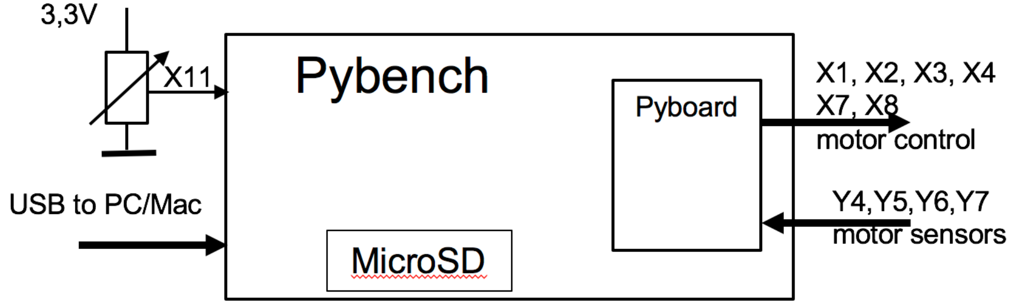


Driving the motor with TB6612

```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)
```



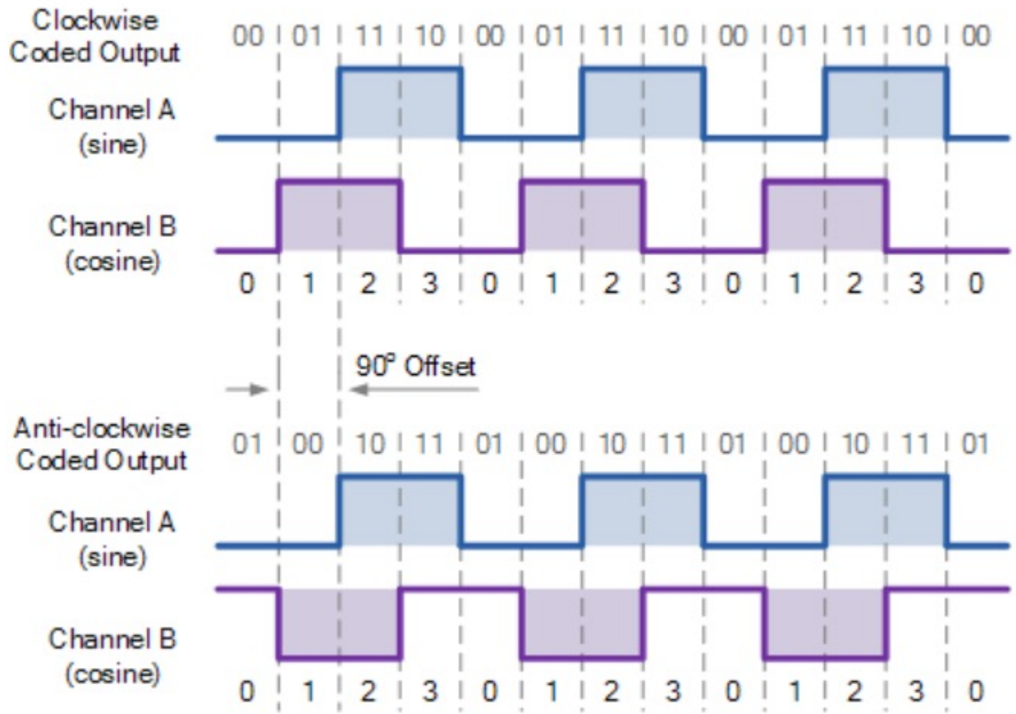
Controlling the speed with potentiometer




```
pot = pyb.ADC(Pin('X11'))        # define potentiometer object as ADC conversion on X11
value = pot.read()               # value = 0 to 4095 for voltage 0v to 3.3v
```

```
while True:                  # loop forever until CTRL-C
    speed = int((pot.read()-2048)*200/4096)
    if (speed >= 0):             # forward
        A_forward(speed)
        B_forward(speed)
    else:
        A_back(abs(speed))
        B_back(abs(speed))
```

Measuring Motor speed with Hall Effect Sensors



- Circular magnet has 13 pole pairs
- The gearbox of the motor has a 1:30 gear ratio
- How many pulses are produced for each revolution of the motor?
- Speed of motor (in rps) can be measured by counting the number of pulses in a given time window (say 100msec)

```
# Define pins for motor speed sensors
A_sense = Pin('Y4', Pin.PULL_NONE)  #  Pin.PULL_NONE = leave this as input pin
B_sense = Pin('Y6', Pin.PULL_NONE)
```

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
```

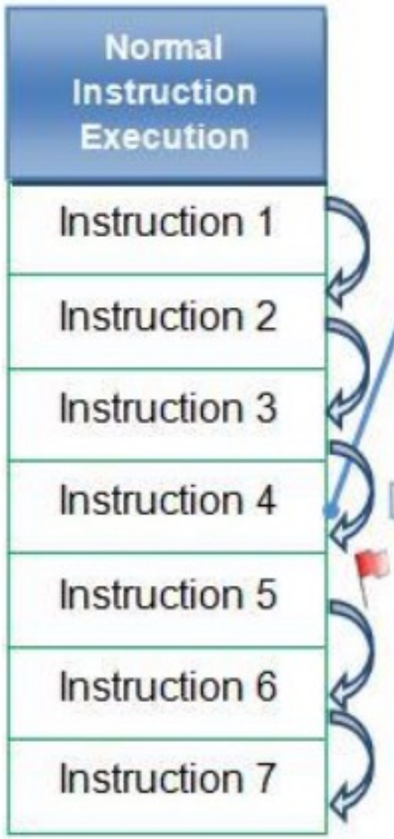
Discuss: what is the limitation of polling?

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

```
# 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()
```

Lab 5: The idea of interrupt



Interrupt occurs while in instruction 4

1. Save the state of program
2. **Jump to ISR**
3. **Stop further interrupts**
4. When finish return

- Hardware method to detect event (e.g. rising edge on a pin), generate interrupt
- Processor forced to do something else – defined in the Interrupt Service Routine (ISR)
- Return when finished

Lab 5: Interrupt Service Routines

- Need to detect and handle two types of events:
  1. Rising edge on Hall effect sensor signal on Y4
  2. 100ms elapsed time on a Timer
- Need two ISRs for these two interrupt events
- Need to provide a dummy variable as shown here

```python
#-------  Section to set up Interrupts ----------
def isr_motorA(dummy):  # motor sensor ISR - just count transitions
    global A_count
    A_count += 1

def isr_speed_timer(dummy):      # timer interrupt at 100msec intervals
    global A_count
    global A_speed
    A_speed = A_count            # remember count value
    A_count = 0                  # reset the count
```

Lab 5: 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)
```

Lab 5 – Interrupt MAGIC

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

    # drive motor - controlled by potentiometer
    speed = int((pot.read()-2048)*200/4096)
    if (speed >= 0):        # forward
        A_forward(speed)
        B_forward(speed)
    else:
        A_back(abs(speed))
        B_back(abs(speed))

    # 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 assumes A_speed has the correct value!
- There is no reference to 100ms time window, nor counting of edges.

Three Big Ideas

1. **PWM** is the efficient way to drive **motors** or **LEDs**. The **H-bridge** motor driver allows PWM signal to control the speed with separate digital signals to control the direction of the motor.
2. **Interrupt** is a much better way of detecting hardware events than using **polling** method.
3. Interrupt makes software hard to debug because once set up, it runs in the background all the time and is difficult to stop. So **make interrupt service routine** as **simple** as possible.