Final

Deliverablecode

Team ID	PNT2022TMID45392
Project Name	Smart Waste Management In Metropolitan

Submitted by:

Team Leader : ARUL JOTHIKA S
Team member : EVANGELINE S
Team member : BRINDHA R
Team member : DEBORAH V

Code:

MAIN.PY

```
C=1
        import time
         for i in range(1,2):
         while True:
            distance == "90 %":if c == 1:
            import distance
            d=distance.distancesensor()
            c = 2
           elif c == 2:
           import load
           w = int(load.loop())
           c = 3
           else:
           import database as db
           if w < 5000 and w > 4000:
           load = "90 %"
           elif w < 4000 and w > 3000:
           load = "60 %"
           elif w < 3000 and w > 100:
           load = "40 %"
           else:
           load = "0 %"
           if d > 30:
           distance = "90 %"
           elif d < 30 and d >20:
           distance = "60 %"
           elif d < 20 and d > 5:
          distance = "40 %"
          else:
          distance = "7 %"
          if load == "90 %" or
         m = "Risk Warning: Dumpster poundage getting high,
Time                                    to collect :)"
         elif load == "60 %" or distance == "60 %":
         m ="dumpster is above 60%"
         else :
         m = " "
         db.database(d,w,m,load,distance)
         print("data pushed")
```

```
          c = 1
         LOAD.py
import
time
         import sys
        EMULATE_HX711=False
        referenceUnit = 1
        if not EMULATE_HX711:
        import RPi.GPIO as GPIO
        from hx711 import HX711
        else:
        from emulated_hx711 import HX711
       def cleanAndExit():
       print("Cleaning...")
       If not EMULATE_HX711:
       GPIO.cleanup()
       print("Bye!")
       sys.exit()
       hx = HX711(5, 6)
           # I've found out that, for some reason, the order of the bytes is not always the
same between versions of python,
numpy and the hx711 itself.
          # Still need to figure out why does it change.
         # If you're experiencing super random values, change these values to MSB or LSB
until to get more stable values.
         # There is some code below to debug and log the order of the bits and the bytes.
         # The first parameter is the order in which the bytes are used to build the "long"
value.
         # The second paramter is the order of the bits inside each byte.
        # According to the HX711 Datasheet, the second parameter is MSB so you
shouldn't need to modify it.
hx.set_reading_format("MSB", "MSB")

        # HOW TO CALCULATE THE REFFERENCE UNIT
        # To set the reference unit to 1. Put 1kg on your sensor or anything you have and
know exactly how much it
weights.
        # In this case, 92 is 1 gram because, with 1 as a reference unit I got numbers near
0 without any weight
       # and I got numbers around 184000 when I added 2kg. So, according to the rule of
thirds:
      # If 2000 grams is 184000 then 1000 grams is 184000 / 2000 = 92.
      hx.set_reference_unit(113)
     #hx.set_reference_unit(referenceUnit)
     hx.reset()
     hx.tare()
     print("Tare done! Add weight now...")
     # to use both channels, you'll need to tare them both
     #hx.tare_A()
    #hx.tare_B()
     def loop():
     try:
     # These three lines are usefull to debug wether to use MSB or LSB in the reading
formats
     # for the first parameter of "hx.set_reading_format("LSB", "MSB")".
    # Comment the two lines "val = hx.get_weight(5)" and "print val" and uncomment
these three lines to see what
```

```
    it prints.
    # np_arr8_string = hx.get_np_arr8_string()
    # binary_string = hx.get_binary_string()
    # print binary_string + " " + np_arr8_string
    # Prints the weight. Comment if you're debbuging the MSB and LSB issue.
     val = hx.get_weight(5)
     print(val)
     return val
    # To get weight from both channels (if you have load cells hooked up
   # to both channel A and B), do something like this
    #val_A = hx.get_weight_A(5)
    #val_B = hx.get_weight_B(5)
    #print "A: %s B: %s" % ( val_A, val_B )

    hx.power_down()
    hx.power_up()
    time.sleep(0.1)
    except (KeyboardInterrupt, SystemExit):
    cleanAndExit()
    DISTANCE.py
    import RPi.GPIO as GPIO
    import time
   def distancesensor():
   try:
   GPIO.setmode(GPIO.BOARD)
   GPIO.setwarnings(False)
   PIN_TRIGGER = 23
   PIN_ECHO = 33
   GPIO.setup(PIN_TRIGGER, GPIO.OUT)
   GPIO.setup(PIN_ECHO, GPIO.IN)
   GPIO.output(PIN_TRIGGER, GPIO.LOW)
   time.sleep(2)
   GPIO.output(PIN_TRIGGER, GPIO.HIGH)
    time.sleep(0.00001)
    GPIO.output(PIN_TRIGGER, GPIO.LOW)
    while GPIO.input(PIN_ECHO)==0:
    pulse_start_time = time.time()
    while GPIO.input(PIN_ECHO)==1:
    pulse_end_time = time.time()
    pulse_duration = pulse_end_time - pulse_start_time
    global distance
   distance = round(pulse_duration * 17150, 2)
   print(distance)
   return distance
    finally:
    GPIO.cleanup
HX711.py import RPi.GPIO as
GPIO
import time
import threading
class HX711:
def init (self, dout, pd_sck, gain=128):
self.PD_SCK = pd_sck
self.DOUT = dout
# Mutex for reading from the HX711, in case multiple threads in
client
# software try to access get values from the class at the same time.
```

```
self.readLock = threading.Lock()
GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)
GPIO.setup(self.PD_SCK, GPIO.OUT)
GPIO.setup(self.DOUT, GPIO.IN)
self.GAIN = 0
# The value returned by the hx711 that corresponds to your
reference
# unit AFTER dividing by the SCALE.
self.REFERENCE_UNIT = 1
self.REFERENCE_UNIT_B = 1
self.OFFSET = 1
self.OFFSET_B = 1
self.lastVal = int(0)
self.DEBUG_PRINTING = False
self.byte_format = 'MSB'
self.bit_format = 'MSB'
self.set_gain(gain)

# Think about whether this is necessary.
time.sleep(1)
def convertFromTwosComplement24bit(self, inputValue):
return -(inputValue & 0x800000) + (inputValue & 0x7fffff)
def is_ready(self):
return GPIO.input(self.DOUT) == 0
def set_gain(self, gain):
if gain is 128:
self.GAIN = 1
elif gain is 64:
self.GAIN = 3
elif gain is 32:
self.GAIN = 2
GPIO.output(self.PD_SCK, False)
# Read out a set of raw bytes and throw it away.
self.readRawBytes()
def get_gain(self):
if self.GAIN == 1:
return 128
if self.GAIN == 3:
return 64
if self.GAIN == 2:
return 32
# Shouldn't get here.
return 0
def readNextBit(self):
# Clock HX711 Digital Serial Clock (PD_SCK). DOUT will be
# ready 1us after PD_SCK rising edge, so we sample after
# lowering PD_SCL, when we know DOUT will be stable.
GPIO.output(self.PD_SCK, True)
GPIO.output(self.PD_SCK, False)
value = GPIO.input(self.DOUT)
# Convert Boolean to int and return it.
return int(value)
def readNextByte(self):
byteValue = 0
# Read bits and build the byte from top, or bottom, depending
# on whether we are in MSB or LSB bit mode.
```

```
for x in range(8):
if self.bit_format == 'MSB':
byteValue <<= 1
byteValue |= self.readNextBit()
else:
byteValue >>= 1
byteValue |= self.readNextBit() * 0x80
# Return the packed byte.
return byteValue
def readRawBytes(self):
# Wait for and get the Read Lock, incase another thread is already
# driving the HX711 serial interface.
self.readLock.acquire()
# Wait until HX711 is ready for us to read a sample.
while not self.is_ready():
pass
# Read three bytes of data from the HX711.
firstByte = self.readNextByte()
secondByte = self.readNextByte()
thirdByte = self.readNextByte()
# HX711 Channel and gain factor are set by number of bits read
# after 24 data bits.
for i in range(self.GAIN):
# Clock a bit out of the HX711 and throw it away.
self.readNextBit()
# Release the Read Lock, now that we've finished driving the
HX711
# serial interface.
self.readLock.release()
# Depending on how we're configured, return an orderd list of raw
byte
# values.
if self.byte_format == 'LSB':
return [thirdByte, secondByte, firstByte]
else:
return [firstByte, secondByte, thirdByte]
def read_long(self):
# Get a sample from the HX711 in the form of raw bytes.
dataBytes = self.readRawBytes()
if self.DEBUG_PRINTING:
print(dataBytes,)
# Join the raw bytes into a single 24bit 2s complement value.
twosComplementValue = ((dataBytes[0] << 16) |
(dataBytes[1] << 8) |
dataBytes[2])
if self.DEBUG_PRINTING:
print("Twos: 0x%06x" % twosComplementValue)
# Convert from 24bit twos-complement to a signed value.
signedIntValue =
self.convertFromTwosComplement24bit(twosComplementValue)
# Record the latest sample value we've read.
self.lastVal = signedIntValue
# Return the sample value we've read from the HX711.
return int(signedIntValue)
def read_average(self, times=3):
# Make sure we've been asked to take a rational amount ofsamples.
if times <= 0:
```

```
raise ValueError("HX711()::read_average(): times must >= 1!!")
# If we're only average across one value, just read it and return it.
if times == 1:
return self.read_long()
# If we're averaging across a low amount of values, just take the
# median.
if times < 5:
return self.read_median(times)
# If we're taking a lot of samples, we'll collect them in a list,
remove
# the outliers, then take the mean of the remaining set.
valueList = []
for x in range(times):
valueList += [self.read_long()]
valueList.sort()
# We'll be trimming 20% of outlier samples from top and bottom
of collected set.
trimAmount = int(len(valueList) * 0.2)
# Trim the edge case values.
valueList = valueList[trimAmount:-trimAmount]
```