```
#

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.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 of samples.
    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]

    # Return the mean of remaining samples.
    Return sum(valueList) / len(valueList)


  # A median-based read method, might help when getting random value spikes
  # for unknown or CPU-related reasons
  Def read_median(self, times=3):
    If times <= 0:
```

```
    Raise ValueError("HX711::read_median(): times must be greater than zero!")

  # If times == 1, just return a single reading.
  If times == 1:
    Return self.read_long()

  valueList = []

  for x in range(times):
    valueList += [self.read_long()]

  valueList.sort()

  # If times is odd we can just take the centre value.
  If (times & 0x1) == 0x1:
    Return valueList[len(valueList) // 2]
  Else:
    # If times is even we have to take the arithmetic mean of
    # the two middle values.
    Midpoint = len(valueList) / 2
    Return sum(valueList[midpoint:midpoint+2]) / 2.0


# Compatibility function, uses channel A version
Def get_value(self, times=3):
   Return self.get_value_A(times)


Def get_value_A(self, times=3):
```

```
    Return self.read_median(times) – self.get_offset_A()



Def get_value_B(self, times=3):
    # for channel B, we need to set_gain(32)
    G = self.get_gain()
    Self.set_gain(32)
    Value = self.read_median(times) – self.get_offset_B()
    Self.set_gain(g)
    Return value

# Compatibility function, uses channel A version
Def get_weight(self, times=3):
    Return self.get_weight_A(times)



Def get_weight_A(self, times=3):
    Value = self.get_value_A(times)
    Value = value / self.REFERENCE_UNIT
    Return value

Def get_weight_B(self, times=3):
    Value = self.get_value_B(times)
    Value = value / self.REFERENCE_UNIT_B
    Return value



# Sets tare for channel A for compatibility purposes
Def tare(self, times=15):
```

```
    Return self.tare_A(times)


Def tare_A(self, times=15):
    # Backup REFERENCE_UNIT value
    backupReferenceUnit = self.get_reference_unit_A()
    self.set_reference_unit_A(1)

    value = self.read_average(times)

    if self.DEBUG_PRINTING:
        print("Tare A value:", value)

    self.set_offset_A(value)

    # Restore the reference unit, now that we've got our offset.
    Self.set_reference_unit_A(backupReferenceUnit)

    Return value


Def tare_B(self, times=15):
    # Backup REFERENCE_UNIT value
    backupReferenceUnit = self.get_reference_unit_B()
    self.set_reference_unit_B(1)

    # for channel B, we need to set_gain(32)
    backupGain = self.get_gain()
    self.set_gain(32)
```

```
    value = self.read_average(times)

    if self.DEBUG_PRINTING:
      print(“Tare B value:”, value)

    self.set_offset_B(value)

    # Restore gain/channel/reference unit settings.
    Self.set_gain(backupGain)
    Self.set_reference_unit_B(backupReferenceUnit)

    Return value


  Def set_reading_format(self, byte_format=”LSB”, bit_format=”MSB”):
    If byte_format == “LSB”:
      Self.byte_format = byte_format
    Elif byte_format == “MSB”:
      Self.byte_format = byte_format
    Else:
      Raise ValueError(“Unrecognised byte_format: \”%s\”” % byte_format)

    If bit_format == “LSB”:
      Self.bit_format = bit_format
    Elif bit_format == “MSB”:
      Self.bit_format = bit_format
    Else:
```

```
      Raise ValueError(“Unrecognised bitformat: \”%s\”” % bit_format)




# sets offset for channel A for compatibility reasons
Def set_offset(self, offset):
   Self.set_offset_A(offset)

Def set_offset_A(self, offset):
   Self.OFFSET = offset

Def set_offset_B(self, offset):
   Self.OFFSET_B = offset

Def get_offset(self):
   Return self.get_offset_A()

Def get_offset_A(self):
   Return self.OFFSET

Def get_offset_B(self):
   Return self.OFFSET_B



Def set_reference_unit(self, reference_unit):
   Self.set_reference_unit_A(reference_unit)
```

```
Def set_reference_unit_A(self, reference_unit):
   # Make sure we aren't asked to use an invalid reference unit.
   If reference_unit == 0:
      Raise ValueError("HX711::set_reference_unit_A() can't accept 0 as a reference unit!")
      Return

   Self.REFERENCE_UNIT = reference_unit


Def set_reference_unit_B(self, reference_unit):
   # Make sure we aren't asked to use an invalid reference unit.
   If reference_unit == 0:
      Raise ValueError("HX711::set_reference_unit_A() can't accept 0 as a reference unit!")
      Return

   Self.REFERENCE_UNIT_B = reference_unit


Def get_reference_unit(self):
   Return get_reference_unit_A()


Def get_reference_unit_A(self):
   Return self.REFERENCE_UNIT


Def get_reference_unit_B(self):
   Return self.REFERENCE_UNIT_B
```

```
Def power_down(self):
   # Wait for and get the Read Lock, incase another thread is already
   # driving the HX711 serial interface.
   Self.readLock.acquire()

   # Cause a rising edge on HX711 Digital Serial Clock (PD_SCK).  We then
   # leave it held up and wait 100 us.  After 60us the HX711 should be
   # powered down.
   GPIO.output(self.PD_SCK, False)
   GPIO.output(self.PD_SCK, True)

   Time.sleep(0.0001)

   # Release the Read Lock, now that we’ve finished driving the HX711
   # serial interface.
   Self.readLock.release()


Def power_up(self):
   # Wait for and get the Read Lock, incase another thread is already
   # driving the HX711 serial interface.
   Self.readLock.acquire()

   # Lower the HX711 Digital Serial Clock (PD_SCK) line.
   GPIO.output(self.PD_SCK, False)

   # Wait 100 us for the HX711 to power back up.
```

```
        Time.sleep(0.0001)

        # Release the Read Lock, now that we've finished driving the HX711
        # serial interface.
        Self.readLock.release()

        # HX711 will now be defaulted to Channel A with gain of 128.  If this
        # isn't what client software has requested from us, take a sample and
        # throw it away, so that next sample from the HX711 will be from the
        # correct channel/gain.
        If self.get_gain() != 128:
          Self.readRawBytes()


    Def reset(self):
        Self.power_down()
        Self.power_up()


# EOF – hx711.py
```