|  |
| --- |
|  |
| 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 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() |
|  |  |