**Amr’s Code - Reviewed by Dominik, Derek, and Irusha**

from \_\_future\_\_ import division

import spidev

import time

import threading

\_\_author\_\_ = "AmrGawish"

\_\_date\_\_ = "$Nov 20, 2015 3:43:40 PM$"

READ\_THREAD = 0

SEND\_THREAD = 1

SAMPLE\_PERIOD\_IN\_SEC = 30

SECONDS\_PER\_SEND = 1

HUMAN\_BPM\_LIMIT = 150

SECONDS\_PER\_MIN = 60

THRESHOLD = 0.65

bt\_index = 0

min\_seconds\_per\_beat = SECONDS\_PER\_MIN/HUMAN\_BPM\_LIMIT

flag = [False, False]

turn = READ\_THREAD

def bitstring(n):

s=bin(n)[2:]

return '0'\*(8-len(s))+s

#Function to read a value from the pulse sensor attached through the ADC coverter

def read (adc\_channel=0 , spi\_channel=0):

conn = spidev.SpiDev(0,spi\_channel)

conn.max\_speed\_hz = 1200000

conn.mode = 0

cmd =192

if adc\_channel:

cmd+=32

reply\_bytes= conn.xfer2([cmd,0])

reply\_bitstring = ''.join(bitstring(n) for n in reply\_bytes)

reply = reply\_bitstring[5:15]

conn.close()

return int(reply,2)/ 2\*\*10

def send(BPM):

print BPM

def sender\_thread(beat\_times, n):

BPM\_array = []

count\_of\_BPM = 0

while(True):

time.sleep(SECONDS\_PER\_SEND)

temp = calculate\_average\_bpm(beat\_times)

BPM\_array.append(temp)

send(temp)

count\_of\_BPM = count\_of\_BPM + 1

def reader\_thread(beat\_times, n):

while True:

voltage = read()

if voltage > THRESHOLD:

#print voltage

#flag[READ\_THREAD] = True

#turn = SEND\_THREAD

#while flag[SEND\_THREAD] and turn == SEND\_THREAD:

# pass

sync\_pre\_cs(READ\_THREAD, SEND\_THREAD)

#CS start

beat\_times.append(time.time())

#CS end

flag[READ\_THREAD] = False

time.sleep(min\_seconds\_per\_beat)

def sync\_pre\_cs(this\_thread, other\_thread):

flag[this\_thread] = True

turn = other\_thread

while flag[other\_thread] and turn == other\_thread:

pass

def calculate\_average\_bpm(beat\_times):

old\_beat\_times = []

ref\_time = time.time()

#flag[SEND\_THREAD] = True

#turn = READ\_THREAD

#while flag[READ\_THREAD] and turn == READ\_THREAD:

# pass

sync\_pre\_cs(SEND\_THREAD, READ\_THREAD)

#CS start

for b\_time in beat\_times:

if(abs(ref\_time - b\_time) > SAMPLE\_PERIOD\_IN\_SEC):

old\_beat\_times.append(b\_time)

for b\_time in old\_beat\_times:

beat\_times.remove(b\_time)

length = len(beat\_times)

#CS end

flag[SEND\_THREAD] = False

if length <= 1 :

return 0

return length\*SECONDS\_PER\_MIN/(beat\_times[length - 1] - beat\_times[0])

threads = []

if \_\_name\_\_ == "\_\_main\_\_":

beat\_times = []

num = 0

reader = threading.Thread(target = reader\_thread, args=(beat\_times, num))

sender = threading.Thread(target=sender\_thread, args=(beat\_times, num))

threads.append(reader)

threads.append(sender)

reader.start()

sender.start()

* Functionality
  + Overall functionality works according to the project specifications
  + Input and Output network communications conform to established data protocol
  + Code straightforward to understand
  + Sensors working in the right way
  + No sections of code incomplete
* Comments
  + Meta data at the top (Name, Date Modified)
  + Comments are comprehensible and add something to the maintainability of the code
  + Comments are neither too numerous nor verbose
  + Comments are in the right place and they are useful
* Code smells
  + Tabs and brackets are consistent
  + Code as modular as possible
  + Repetitive code has been factored out
  + Command classes have been designed to undertake one task only
  + The code does not use unjustifiable static methods/blocks
  + Loops have a set length and correct termination conditions
  + Any unusual behavior or edge-case handling described
  + No hardcoded or Magic numbers present within reason
* Performance
  + No unnecessary loops
  + No possible replacement of recursive functions with sequential functions
  + the code was designed to perform as fast as possible
* Scope
  + No possible replacement of global variables to function variables
  + Variable types have been generalized where possible
  + Down casting used properly
* Unit Tests
  + Unit tests are present and correct
  + The code is unit testable
  + Testing the part that is implemented and checked for right performance
* Error/Exception handling
  + Common errors have been checked for
  + No zombie threads running
  + Any security concerns have been addressed
  + data goes through many filters that makes sure the data does not contain errors
* Logging/Debugging Info
  + Logging used appropriately (proper logging level and details)
  + Are all data inputs checked (for the correct type, length, format, and range) and encoded?
* Code Library Usage
  + Frameworks have been used appropriately
  + Can any of the code be replaced with library functions?
  + the code was designed using the libraries and there is no code to be replaced