Dyson School of Design Engineering

Imperial College London

DE2.3 Electronics 2

Lab Experiment 7: Basic Beat Detection implementation

(webpage: http://www.ee.ic.ac.uk/pcheung/teaching/DE2_EE/)

INTRODUCTION

In order assist you to make progress on the project, I here provide you with a skeleton program "**beat_detect_0.py**" for real-time beat detection running on Pybench (in MicroPython). This program works reasonably well for the music "Staying Alive". Your job is to try to improve this basic program to obtain a better performing one. This Lab session should help you in achieving milestone 2 and 3.

CONTEXT

Our goal is to run **real-time code** in MicroPython using **Pybench** to detect when a beat occurs. In this version, the blue LED is flashed whenever a beat is detected. You can substitute flashing the LED with a dancing step (or do both!).

Debugging interrupt driven program is difficult. I found that some students are struggling to get a basic version of the code running without error on **Pybench**. Given the number of deadlines you have, I decided to provide you with various "basic" version of code from which you can learn. Your challenge is to make my implementation better.

You can download this program from the course webpage.

EXPLANATION

```
19  import pyb
20  from pyb import Pin, Timer, ADC, DAC, LED
21  from array import array         # need this for memory allocation to buffers
22  from oled_938 import OLED_938   # Use OLED display driver
23
24  #  The following two lines are needed by micropython
25  #   ... must include if you use interrupt in your program
26  import micropython
27  micropython.alloc_emergency_exception_buf(100)
28
29  # I2C connected to Y9, Y10 (I2C bus 2) and Y11 is reset low active
30  oled = OLED_938(pinout={'sda': 'Y10', 'scl': 'Y9', 'res': 'Y8'}, height=64,
31                  external_vcc=False, i2c_devid=61)
32  oled.poweron()
33  oled.init_display()
34  oled.draw_text(0,0, 'Basic Beat Detection')
35  oled.display()
36
37  # define ports for microphone, LEDs and trigger out (X5)
38  mic = ADC(Pin('Y11'))
39  MIC_OFFSET = 1523       # ADC reading of microphone for silence
40  b_LED = LED(4)      # flash for beats on blue LED
```

Lines 19-22: import packages and classes used.

Lines 26,27: Micropython requires this if you use interrupts anywhere in your program.

Lines 29-40: Construct various hardware objects used later in this program.

MIC_OFFSET is the microphone reading from the ADC when it is quiet. ADC converts 0 – 3.3V range to 0 – 4095 (12-bit ADC). Microphone amplifier is at a voltage that converts to 1523 on my Pybench.

You may have a different offset value for your Pybench.

```
42  N = 160             # size of sample buffer s_buf[]
43  s_buf = array('H', 0 for i in range(N))  # reserve buffer memory
44  ptr = 0             # sample buffer index pointer
45  buffer_full = False # semaphore - ISR communicate with main program
46
47  def flash():        # routine to flash blue LED when beat detected
48      b_LED.on()
49      pyb.delay(20)
50      b_LED.off()
51
52  def energy(buf):    # Compute energy of signal in buffer
53      sum = 0
54      for i in range(len(buf)):
55          s = buf[i] - MIC_OFFSET # adjust sample to remove dc offset
56          sum = sum + s*s         # accumulate sum of energy
57      return sum
58
59  # ---- The following section handles interrupts for sampling data -----
60  # Interrupt service routine to fill sample buffer s_buf
61  def isr_sampling(dummy):    # timer interrupt at 8kHz
62      global ptr              # need to make ptr visible inside ISR
63      global buffer_full      # need to make buffer_full inside ISR
64
65      s_buf[ptr] = mic.read() # take a sample every timer interrupt
66      ptr += 1                # increment buffer pointer (index)
67      if (ptr == N):          # wraparound ptr - goes 0 to N-1
68          ptr = 0
69          buffer_full = True  # set the flag (semaphore) for buffer full
70
71  # Create timer interrupt - one every 1/8000 sec or 125 usec
72  sample_timer = pyb.Timer(7, freq=8000)  # set timer 7 for 8kHz
73  sample_timer.callback(isr_sampling)     # specify interrupt service routine
74  # -------- End of interrupt section ----------------
```

Lines 42: N is the number of samples we acquire in each energy window. 160 sample is equivalent to 20msec window. This is one "*epoch*".

Line 43: This is the way to reserve memory for **s_buf**, the sample buffer. There are **N** locations in the buffer. '**H'** means data format is half integer or 16-bit. Capital H means it is unsigned (ADC returns value 0 – 4095). **s_buf** also initialized to 0.

Lines 47 – 50: Flash blue LED.

Lines 52 – 57: Compute energy in a 20msec epoch. For each sample, we also remove the dc offset first.

Lines 61 – 69: This is the **interrupt service routine** that get executed automatically, once every sample period.

It stores samples in **s_buf**. **ISR** should be as short as possible. **ptr** is the index to **s_buf**, where the next sample should be stored. It goes from 0 to N-1, and get incremented each time the ISR is called. When **ptr** reaches N, it get reset to 0, and the "**buffer_full**" flag is then set to tell the main program loop that we now have N signal samples in **s_buf**.

Line 72: Initialize timer 7 so that its times out every 1/8000 sec. This is used as the sampling clock. Each sampling period is now 127 microseconds.

Line 73: Set up the interrupt for timer 7. Whenever timer 7 times out, i.e. 125 microseconds has elapsed, the routine "**isr_sampling**" is called. Interrupt service routine in Python is called "**callback**" function.

```
76  # Define constants for main program loop - shown in UPPERCASE
77  M = 50                      # number of instantaneous energy epochs to sum
78  BEAT_THRESHOLD = 2.0        # threshold for c to indicate a beat
79
80  # initialise variables for main program loop
81  e_ptr = 0                   # pointer to energy buffer
82  e_buf = array('L', 0 for i in range(M)) # reserve storage for energy buffer
83  sum_energy = 0              # total energy in last 50 epochs
84  tic = pyb.millis()          # mark time now in msec
85
86  while True:             # Main program loop
87      if buffer_full:     # semaphore signal from ISR - set if buffer is full
88
89          # Calculate instantaneous energy
90          E = energy(s_buf)
91
92          # compute moving sum of last 50 energy epochs
93          sum_energy = sum_energy - e_buf[e_ptr] + E
94          e_buf[e_ptr] = E        # over-write earlest energy with most recent
95          e_ptr = (e_ptr + 1) % M # increment e_ptr with wraparound - 0 to M-1
96
97          # Compute ratio of instantaneous energy/average energy
98          c = E*M/sum_energy
99
100         if (pyb.millis()-tic > 500):    # if more than 500ms since last beat
101             if (c>BEAT_THRESHOLD):      # look for a beat
102                 flash()                 # beat found, flash blue LED
103                 tic = pyb.millis()      # reset tic
104         buffer_full = False             # reset status flag
```

Line 77: M is the number of instantaneous energy values to average over to obtain the average local energy.

Line 78: **BEAT_THRESHOLD** is the ratio of *instantaneous energy / local average energy* beyond which a beat is detected.

Line 81: **e_ptr** is the index for a buffer storing M instant energy values.

Line 82: **e_buf** is the instant energy buffer of length M. Data format is a regular unsigned integer. '**L**' is normal integer, i.e. 32-bits, uppercase is unsigned.

Lines 86 – 104: Main program loop. This is what all real-time program would look like. It loops around forever.

Line 87: The time it takes to go around the loop once is determined by the **buffer_full** flag, which is set in the sampling interrupt service routine once the buffer is full. The buffer has N=160 locations, and the sampling period is 1/8000 = 125 μsec. Therefore, the loop goes around once every 20msec.

Line 90: Compute energy in sample buffer – one epoch. This returns the instantaneous energy E.

Line 93: This is a clever trick! We want to find the average energy of the past M instant energy values. We could do this by summing up what's stored in **e_buf[0]** to **e_buf[M-1]**. That takes M-1 adds. However, we can also keep a running sum of instant energy **sum_energy**, take away the earliest instant energy value, and then add the current **E**. This takes only two adds (or subtract) – much quicker!

Line 94: Overwrite the earliest sample in buffer with this new instantaneous energy **E.** **e_ptr** is pointing to (i.e. providing the index for) the oldest sample in **e_buf[].**

Line 95: Update **e_ptr** to move to the next oldest sample, soon to be overwritten. The **"% M**" operation is modulo M (divide by M and get the remainder). It is a method to increment the index value, make sure that this value stay within 0 to M-1, and wrap it around whenever it reaches M. In that way, **e_buf[ ]** will always have the past M instant energy values, and this buffer get updated each echo (20msec) period.

Line 98: Calculate the ratio **c**, *instantaneous energy / average energy*. **sum_energy** has the total energy over 50 epochs. **sum_energy/M** is the average.

Line 100: Check that the elapsed time is 500msec or more since detecting the last beat. We know that "Staying Alive" has a beat period of around 570msec from your MATLAB analysis. So we only expect the next beat 500msec or later.

Line 101: Beat is detected only if *c > some threshold*. Change the threshold will affect accuracy of detection.

Line 104: Reset the **buffer_full** status flag, ready for another 20msec period

WHAT IS NEXT?

To make the mini-Segway dance to music, you would need to have created the dance routine in the form of steps encoded in ASCII characters. The dance routine can be created manually or automatically. You can then replace "**flash()"** with the appropriate function to move the mini-Segway. With the stabilizer installed, your Segway should dance to the music.

For a different song, the beat period would be different. You would need to change the program so that it looks for a beat earlier or later than 500msec in the current basic program.