Lab 5 & 6: Team Project Sessions 1 & 2

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

Introduction

You have finished all formal laboratory sessions for DE2.3. From now, you should focus only on the Team Project during the lab sessions. This document is the combined instruction for two lab sessions. It is intended to offer you some guidance and hints as to how to go about the complex task of achieving the goals of Milestones 1 and 2. Before that, let us do one more exercise on filters.

Exercise 1: Moving Average filter

This exercise will be conducted using Matlab alone. The goal is for you to explore the low pass filtering effect of the moving average filter.

Download the music file: bgs.wav from the course webpage. This is a short segment of music from “Staying Alive”.

Create the following Matlab script as lab5_ex1.m.

```
% Lab 5 - Ex 5a: 4-taps Moving average filter
%    inefficient way
clear all
[sig fs] = audioread('bgs.wav');
% Add noise to music
x = sig + 0.2*rand(size(sig));
% Plot the signal
figure(1);
clf;
plot(x);
xlabel('Sample no');
ylabel('Signal (v)');
title('Stay Alive Music');
% Filter music with moving average filter
N = size(x); %
for i=4:N
    y(1) = (x(1)+x(i-1)+x(i-2)+x(i-3))/4;
end
y(1)=x(1)/4;
y(2)=(x(2)+x(1))/4;
y(3)=(x(3)+x(2)+x(1))/4;
% Play the original & then the filtered sound
sound(x, fs)
disp('Playing the original - press return when finished')
pause;
sound(y, fs)
disp('Playing the filter music')
```

When you compare the noise-corrupted music with the filtered version, you should notice a slight reduction in the noise.
Modify lab5_ex1a.m to lab5_ex1b.m so that you can use a variable number of taps. Change this to 10 and 20, and see how the filtered music sound. (Solution in Appendix A.)

**Milestone 1**

By the end of today, you should have put together your Segway with the stabilizer installed. I have laser cut enough stabilisers for 25 teams, but you are welcome to design your own. A sample of the completed Segway with stabilizer is available for you to inspect.

You also need to write the controlling program in MicroPython to drive the Segway using the AdaFruit mobile App via Bluetooth.

**Sampling microphone signal using interrupt**

Although you are not expected to complete Milestone 2 until the end of February, you should read through the instruction below concerning how to sample and buffer N microphone signals under the control of a timer interrupt. The timer is programmed to issue an interrupt at an 8kHz sampling rate. That is, an interrupt is generated EVERY 125 µsec.

The code to do this is shown below and is stored in the file ‘buffer.py’, which you can download from the course webpage.

This program will store away N samples in a pre-allocated array ‘s_buf[,]’ in the interrupt service routine ‘isr_sampling[,]’. Once set up, this will happen automatically in the background without further instruction from the main program loop in the foreground.

We also need a way to tell the main program loop that the buffer is full, i.e. N samples have been taken. To do this, we use a status flag (also known as a ‘semaphore’) to indicate that N samples have been taken.

The main program loop simply wait until the buffer is filled, and then it displays the buffer content on the OLED display the function ‘plot_signal[,]’.

Finally, we have created an output pin object on ‘X5’, which is connected to the top BNC connector (used as analogue output in earlier experiments). We then toggle this pin in the main program loop to indicate the time it takes for the main loop to go around the loop. The time it takes for the semaphore to go high is the half period (i.e. it toggles every N x 125 µsec).

If you now measure the signal on X5, you will see that it is much slower than expected. Why? Because driving the OLED display is slow! If you now comment out the OLED display statements, you will see that X5 signal is as expected.

You should find this buffer program useful when you write the code to synchronise the movement of the Segway to the beat of the music.

**Milestone 2 (leading to 3)**

For milestone 2, you should program the Segway to dance to music WITHOUT balancing (because the stabilizer will keep the Segway upright). Before you attempt Milestone 2, pick a piece of music that has a strong beat. Your Segway will dance to this music in real-time. Two years ago, all groups were told to use “Staying Alive” by the Bee Gees. This piece has very clear beat and is useful to test your algorithm. However, you are encouraged to choose your own.

Milestone 2 is a relatively large task, which can be broken down into the following sub-tasks:

1) Process the music signal offline using Matlab to derive the beat period (i.e. how long between beats and therefore the duration of each dancing step);
2) Determine the colour, mood, shape of the music through the spectrum of around a few seconds of the signal (in Matlab);

3) Either automatically map the spectrum of music to dancing steps, or manually decide on the dancing steps manually. Store the step sequence as ASCII text file to be transferred to the microSD card;

4) Write the MicroPython program on PyBench to synchronize the Segway’s movement to the beat of music. There are a few possible approaches to this, some easy, and some much harder. This is discussed below. Once a beat is detected, flash one or more of the four LEDs. You can therefore test your music analysis and the dance routine WITHOUT anything moving!

For the synchronization task, you may consider the following methods:

a) Simply wait until you detect the first beat. Since you have worked out the exact beat period, and if ‘Staying Alive’ has a steady beat, you can use

   \[
   \text{tic} = \text{pyb.millis}() \\
   \text{...} \\
   \text{elapsed} \_ \text{time} = \text{pyb.millis}() - \text{tic}
   \]

   to determine exactly when the next beat is. The disadvantage is that timing error will accumulate. However, you will probably be approximately right, particularly at the beginning.

b) You can find the ratio of instantaneous energy over a 20 msec window to the local average energy over 1 or 2 second moving window, and detect the start of the beat when this ratio exceeds some threshold.

c) To reduce the chance of false beat detection, you can look for the beat only after a certain period of time has elapsed since the last detected beat.
Beat Detection and flashing the BLUE LED on the Pyboard

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

Lines 19-22: import packages and classes used.

```python
import pyb
from pyb import Pin, Timer, ADC, DAC, LED
from array import array
from oled_938 import OLED_938  # Use OLED display driver
from pyb import Pin, Timer, ADC, DAC, LED
from array import array
from oled_938 import OLED_938  # Use OLED display driver
# The following two lines are needed by micropython
# ... must include if you use interrupt in your program
import micropython
micropython.alloc_emergency_exception_buf(100)
# I2C connected to Y9, Y10 (I2C bus 2) and Y11 is reset low active
oled = OLED_938(sda='Y10', scl='Y9', res='Y8', height=64,
                external_vcc=False, i2c_devid=61)
oled.poweron()
oled.init_display()
oled.draw_text(0, 0, 'Basic Beat Detection')
oled.display()

# define ports for microphone, LEDs and trigger out (XS)
mic = ADC(Pin('Y11'))

MIC_OFFSET = 1523  # ADC reading of microphone for silence
b_LED = LED(4)      # flash for beats on blue LED
```

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.
N = 160  # size of sample buffer s_buf
s_buf = array('H', [0 for i in range(N)])  # reserve buffer memory
ptr = 0  # sample buffer index pointer
buffer_full = False  # semaphore - ISR communicate with main program

def flash():  # routine to flash blue LED when beat detected
    b_LED.on()
    pyb.delay(20)
    b_LED.off()

def energy(buf):  # Compute energy of signal in buffer
    sum = 0
    for i in range(len(buf)):
        s = buf[i] - MIC_OFFSET  # adjust sample to remove dc offset
        sum = sum + s*s  # accumulate sum of energy
    return sum

# --- The following section handles interrupts for sampling data ---
def isr_sampling(dummy):  # timer interrupt
    global ptr  # need to make ptr visible inside ISR
    global buffer_full  # need to make buffer_full inside ISR

    s_buf[ptr] = mic.read()  # take a sample every timer interrupt
    ptr += 1  # increment buffer pointer (index)
    if (ptr == N):  # wraparound ptr - goes 0 to N-1
        ptr = 0
        buffer_full = True  # set the flag (semaphore) for buffer full

    # Create timer interrupt - one every 1/8000 sec or 125 usec
    sample_timer = pyb.Timer(7, freq=8000)  # set timer 7 for 8kHz

    sample_timer.callback(isr_sampling)  # specify interrupt service routine

# ------ End of interrupt section ---------

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 or 125 usec. 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.
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.

TARGET ACHIEVEMENT BY END OF FEBRUARY

Although your team’s achievement on the Dancing Segway project will be evaluated in the last week Wednesday of the term 18 March, I encourage you to complete the dancing without balancing part of project by the end of February. If you take longer, it is not a problem, but the earlier you complete milestones 1 and 2, the more relaxed you will be towards the end of the term.
Appendix B: Solution to Excise 1b

```matlab
% Lab 5 - Ex 5b: N-taps Moving average filter

clear all
[sig fs] = audioread('bgs.wav');
% Add noise to music
x = sig + 0.2*rand(size(sig));
% Plot the signal
figure(1);
clf;
plot(x);
xlabel('Sample no');
ylabel('Signal (v)');
title('Stay Alive Music');
% Filter music with moving average filter
N = size(x);
N_tap = 20
for i=N_tap:N
    temp = 0;
    for j = 0:N_tap-1
        temp = temp + x(i-j);
    end
    y(i) = temp/N_tap;
end
% Play the original
sound(x, fs)
disp('Playing the original - press return when finished')
pause;
sound(y, fs)
disp('Playing the filter music')
```
Appendix B  – Code for “buffer.py”

```python
import pyb
from pyb import Pin, Timer, ADC, DAC, LED
from array import array
from oled_938 import OLED_938  # Use OLED display driver

# I2C connected to Y9, Y10 (I2C bus 2) and Y11 is reset low active
oled = OLED_938(pinout={'sda': 'Y10', 'scl': 'Y9', 'res': 'Y8'}, height=64, external_vcc=False, i2c_devid=61)
oled.poweron()
oled.init_display()
oled.draw_text(0,0, 'Lab 6 - Exercise 1')
oled.display()

# define ports for microphone in and trigger out
mic = ADC(Pin('Y11'))
trigger = Pin('X5', Pin.OUT_PP)

N = 160  # size of sample buffer
s_buf = array('H', [0 for i in range(N)])  # reserve buffer memory
global ptr
global buffer_full
ptr = 0  # buffer index
buffer_full = False  # semaphore - ISR communicate with main program

# Function to plot data on OLED - used only for diagnosis
def plot_sig(signal,message):
    index = len(signal)
    if index >= 128:
        step = min(index,int(index/128))
    else:
        step = 1
    oled.clear()
oled.draw_text(0,0,message)
x = 127
max_sig = max(max(signal),3000)
min_sig = min(min(signal),1000)
range_sig = max_sig - min_sig
for i in range(0,index,step):
    y = 63 - int((signal[i] - min_sig)*63/range_sig)
oled.set_pixel(x,y,True)
x = x - 1
oled.display()
```

V2.2 - PYK Cheung, 19 & 26 Feb 2020
# Interrupt service routine to fill sample buffer s_buf

def isr_sampling(dummy):
    global ptr
    global buffer_full
    s_buf[ptr] = mic.read()  # take a sample every timer interrupt
    ptr += 1
    if (ptr == N):
        ptr = 0
        buffer_full = True

# Create timer interrupt - one every 1/8000 sec or 125 usec
speed_timer = pyb.Timer(7, freq=8000)
speed_timer.callback(isr_sampling)

while True:
    if buffer_full:  # semaphore signal from ISR to say buffer full
        plot_sig(s_buf, 'Hello World!')
        if (trigger.value()):  # Measure how long it takes on XS - top BNC
            trigger.low()
        else:
            trigger.high()
        buffer_filled = False