Lecture 14

Dancing Segway and Analysis of Musical Signal

Peter Cheung

Dyson School of Design Engineering

URL: www.ee.ic.ac.uk/pcheung/teaching/DE2_EE/
E-mail: p.cheung@imperial.ac.uk

Segway Challenge – Aim and Objective

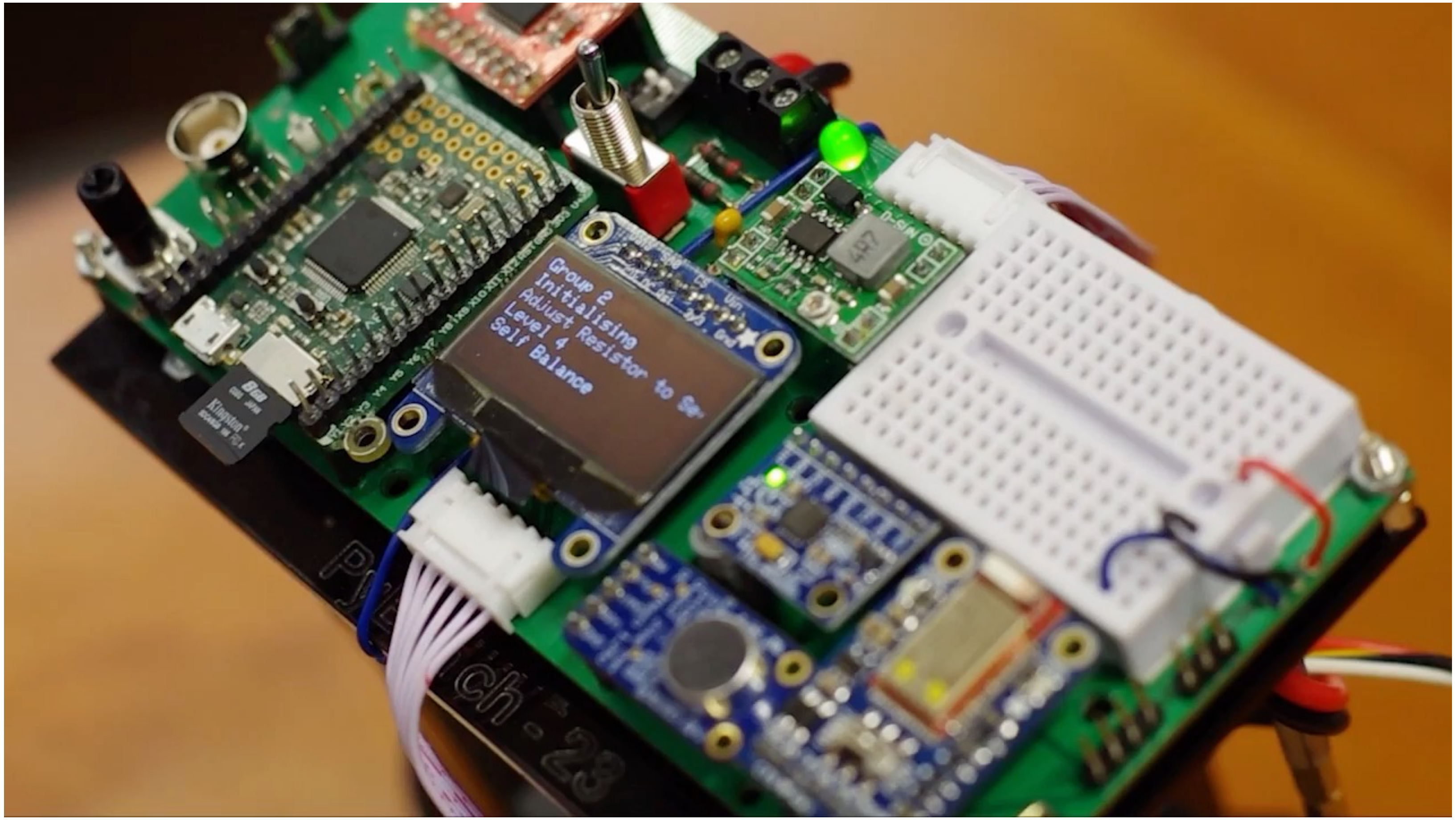
- To demonstrate your understanding of four topics in the Electronics 2 modules that are important to a design engineer:
  1. Signal processing;
  2. System analysis and design;
  3. Feedback control;
  4. Real-time embedded system

- The various challenges are designed to achieve the following:
  1. Apply what you have learned in this module to a real-life problem;
  2. Learn to combine offline processing using Matlab with real-time processing using MicroPython;
  3. Apply embedded system concepts and techniques such as sampling, buffer, interrupts, scheduling etc.;
  4. Have fun!

Segway Project – Learning Outcomes

- By the end of the challenges, you will be able to do most if not all of these:
  1. Process music signals using signal processing techniques to extract its signal characteristics such as rhythm (e.g. beat), ~~spectral contents (e.g. colour) and mood (e.g. swinging, loud, quiet);~~
  2. Creatively map the music characteristics to dance routines (manual);
  3. Analyse music signals in real-time on the microcontroller to synchronize dance movement to music;
  4. Balance a mini-Segway using a PID controller so that it moves around on two wheels under the control of your phone;
  5. Implement the mini-Segway that autonomously dance to live music.

Electronics 2 – from the past!



Capturing real-time audio samples

- Sampling at 8kHz – assume that music signal under 4kHz
- Should use anti-aliasing filter (but not on PyBench)
- To capture the audio signal, you need to:
  1. Set up a timer to produce an interrupt every 125 microsecond
  2. Capture a microphone sample and put it into a buffer s_buf (i.e. an array) which stores N samples in sequence (N is 160 in my code, but can be changed)
  3. When the buffer is full (i.e. N samples capture), set buffer_full to TRUE (this is called a semaphore or a flag)

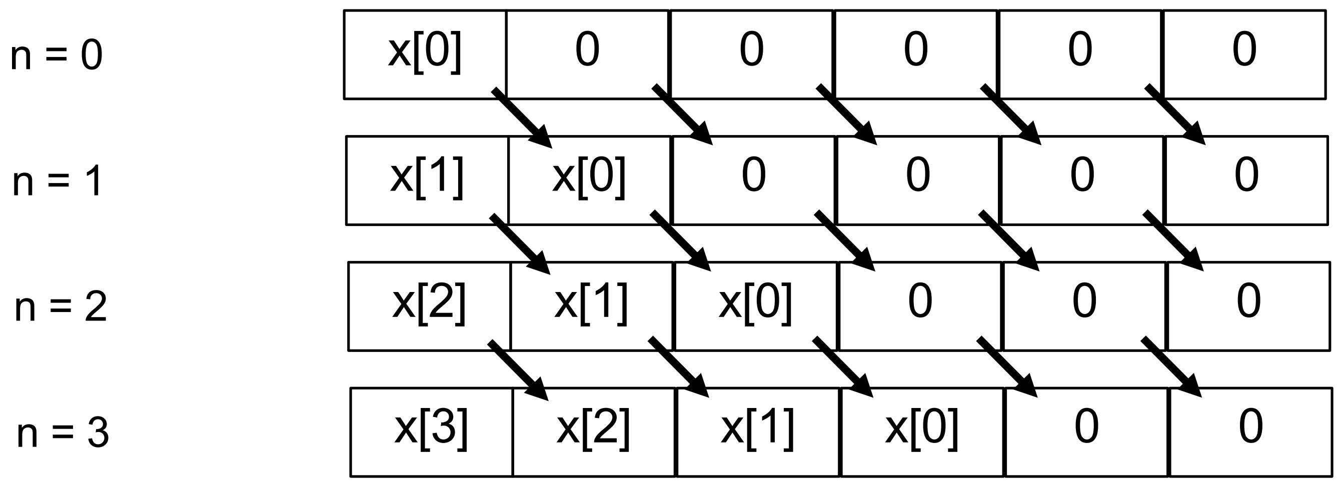
Setting up the Timer to generate an interrupt

- The microcontroller used on Pybench has many timers which can be programmed to produce interrupts
- We will use Timer 7 to generate the sampling interrupt
- Our interrupt service routine (ISR) is **isr_sampling**

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

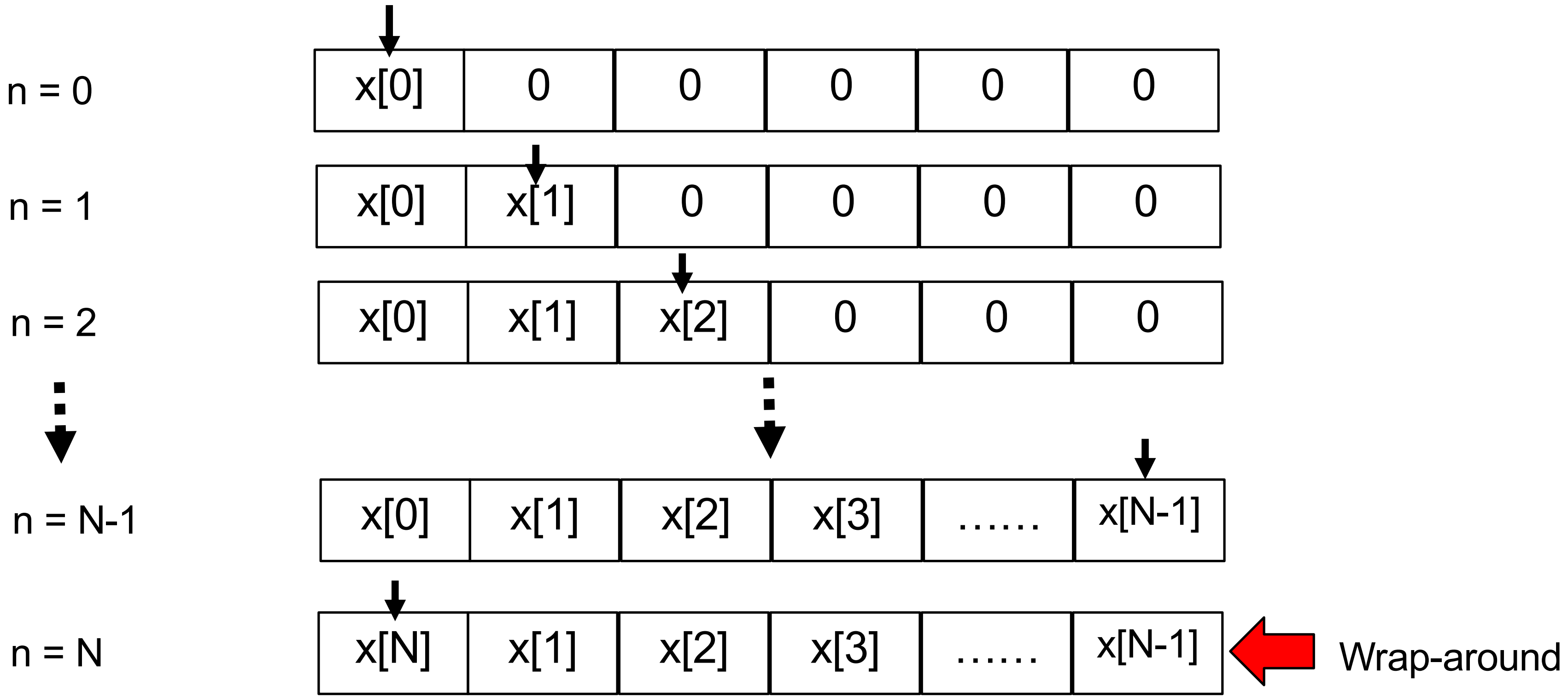
Buffering of signals

- In all the algorithms considered so far, we need to store N data samples. Data could be input music signal (from microphone) x[n], or instantaneous energy $\rho[n]$.
- In Matlab, this is easy. Matlab perform analysis offline, and you can store the signal is a huge array.
- In real-time system, this is not practical (nor possible!).
- Solution: implement a buffer:



Efficient Buffering Method

- Instead of moving lots of data, you can use a "pointer" to specify where to put the new data:
- Use x[ptr], and increment ptr each time a new data comes in.
- Wraparound to 0 when ptr reaches N: `ptr = (ptr + 1) % N`



Interrupt Service Routine - isr_sampling

- The ISR do the following:
  1. Read microphone data
  2. Store it in the next location in array s_buf [ptr] – ptr is the index to the array
  3. Increment index by 1
  4. If index reaches N, buffer is full – set the flag (semaphore)

```python
# Interrupt service routine to fill sample buffer s_buf
def isr_sampling(dummy):     # timer interrupt at 8kHz
    global ptr               # need to make ptr visible in here
    global buffer_full  # need to make buffer_filled visible in here

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

Beat detection using instantaneous energy (method 1)

- Assuming that sampling frequency is 8kHz
- We keep the current sample and N-1 previous samples of input x[n]
- Compute instantaneous energy of sound signal x[n] in, say, 20 msec window (N = 160):

$$\rho[n] = \sum_{k=0}^{159} x[n-k]^2$$

- One approach is to take the Fourier transform of the energy signal ρ[n].
- Collect 1-2 second worth (i.e. 50 to 100 ρ [n] values) and perform FFT on Matlab.
- The fundamental frequency of the spectrum ρ [jω] provides an estimate of the beat frequency.

Beat detection using instantaneous energy (method 2)

- Compute instantaneous energy of sound signal x[n] in 20 msec window:

$$\rho[n] = \sum_{k=0}^{159} x[n-k]^2$$

- Compute steady state local energy by averaging 100 instantaneous energy values $\rho[0]$ to $\rho[99]$:

$$< \rho > \approx \frac{1}{100} \sum_{j=0}^{99} \rho[n-j]$$

- Beat occurs in the window when $\rho[n] > b \times < \rho >$, where b is a threshold chosen for the music.
- Method useful for real-time synchronisation (running MicroPython on Pybench).

Beat detection using instantaneous energy (method 3)

- The problem of the previous method is that if you choose the wrong value for b, the algorithm will not work well.
- The threshold b need to adapt to the music itself. How?
- Compute the variance *v*[n] of the instantaneous energy ρ[n] over 20msec window:

$$v[n] = \frac{1}{100}\sum_{j=0}^{99}(\rho[n-j] - <\rho>)^2$$

- Now computer the threshold value b as:

$$b = \beta - \alpha \times v[n]$$

and try β = 1.5, and α = 0.0025

Beat detection using Frequency selected energy

- Algorithm so far does not consider the frequency content of the music sound. That is, we ignore the frequency spectrum of the signal – it is colour blind!
- We know that beat information in a signal is actually frequency band related.
- Beat from drums – low frequency; beat from cymbal or triangle – high frequency.
- Therefore, assuming that our music is drum heavy, you can low pass filter the signal first before performing the previous beat detection algorithm.

Package to drive motors

- The package **motor.py** is available to help you drive the two motors with ease. It will make developing your milestone code much easier.
- You must first import the package, and then create the motor object:

```python
from motor import DRIVE
# create motor object for the two motors
motor = DRIVE()
```

- Thereafter, you can use the following methods:
- The first five methods are useful to control speed of the motors using the CONTROL PAD via Bluetooth
- The last six methods are directly controlling the movements of the two motors (in an open-loop manner)
- v is not really the speed, but the PWM drive value to the motors.

Method	Description
motor.up_Aspeed(v)	increase motor A speed by v
motor.up_Bspeed(v)	increase motor B speed by v
motor.dn_Aspeed(v)	Reduce motor A speed by v
motor.dn_Bspeed(v)	Reduce motor B speed by v
motor.drive( )	Drive motors at their set speeds
motor.A_forward(v)	Drive motor A forward at v
motor.B_forward(v)	Drive motor B forward at v
motor.A_back(v)	Drive motor A backward at v
motor.B_back(v)	Drive motor B backward at v
motor.A_stop( )	Stop motor A
motor.B_stop( )	Stop motor B