In this lecture, we will introduce the team project – the dancing Segway.

You will be conducting further Lab Sessions main at helping you to achieve the milestones for this project. This lecture will provide some basic background for these lab sessions. In particular, you will learn how to detect beat and handle real-time data capture.
This slide shows the aims and objectives of the team project. As in last year, the project helps you to glue together everything you have learned in this module.

So far, I have covered the necessary theory for aims 1, 2 and 4, either through lectures or through the lab experiments. I will be covering the topic of feedback control starting next lecture.
For any module you take on your course, you should be very clear about learning outcomes. Here is the project’s learning outcomes – stating what you should be able to do when you have finished this project.
**Segway Project – Milestones**

1. **Detect Beats and flash LEDs** (target date: 19th Feb) – To analyse the music using the microphone on the PyBench and detect when the beat occurs. Flash LEDs to indicate beats.

   **Target completion date: 19th Feb.**

2. **Control of the Segway** (target date: 26th Feb) – To fit a pair of “stabilizers” to the two-wheel chassis so that the vehicle will not fall over, but have enough clearance to implement self-balancing later. To drive the Segway with stabilizer along a defined path under the control of the mobile phone connect to the PyBench via Bluetooth.

   **Target completion date: 26th Feb.**

3. **Simple Dance with stabilizer** (target date: 4th Mar) – To analyse the music “Staying Alive” and one other song of your choice in Matlab, create simple dance routines, transfer to the PyBench to store on the SD card and write a Python program on the PyBench to control the Segway with stabilizer so that it moves to music.

   **Target completion date: 4th March.**

4. **“Come Dancing competition”** (18 Mar) – The final assessment will involve demonstrating the dancing Segway that synchronizes to life music, while balancing on two-wheels. Assessment will be based on the robustness of the vehicle, the creativity of the dance routine, the quality of the synchronization to the music etc.

   **Final assessment of the Team will take place on: Wednesday 18th March.**
Here is what milestone 3 could look like. This Segway does not self-balance, but rely on the stabilizer to keep it upright. However, it does synchronise to music and dance!
This is a video showing different groups achievement from a previous year. Some Segways only perform dance to music. Others can do dance and balance at the same time. The final clip shows driving a self-balancing Segway with a camera onboard!
I now want to introduce you to the next Lab. From now on, Labs are not assessed. However, I will be providing you two more instructions to help you with the project.

The goal for Wednesday's lab is to capture audio data in real-time using interrupts, so that you can perform beat detection and eventually synchronize your dance moves to the beat of the music.

---

**Capturing real-time audio samples**

- Sampling at 8kHz – assume that music signal under 4kHz
- Should use anti-aliasing filter (but not on PyBench)
- I provide you with a package: buffer.py to do the sampling of the audio signal from the microphone
- This package do the following:
  1. Set up a timer to produce an interrupt every 125 microseconds
  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)
  4. Update the OLED display to show the captured audio data
This is how you can program a timer (Timer 7) to produce an interrupt every 125 microseconds. The interrupt service routine is specified with the callback function.

Ask yourself this question: what does the microprocessor do when an interrupt occurs?

The answer is:
1. Assuming the microprocessor is enabled to respond to interrupts, it will complete the current instruction;
2. It stores away (in some memory called “stack”) the location of the next instruction, so that it can return later to continue your main program;
3. It stores away the internal values of the processor (called processor state or context);
4. It jumps to the interrupt service routine and do whatever that specifies;
5. On completing the ISR, it recovers the state of the processor (or context);
6. It return to the place in the main program where it was previously interrupted.
We need to take N samples of audio data in order to work out when a beat occurs. To do that, we use something called a buffer.

There is a type of buffer known as FIFO: first-in first-out buffer. This is shown in the diagram below. This is like a queue – queue to get onto a bus.

<table>
<thead>
<tr>
<th>n = 0</th>
<th>x[0]</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>n = 1</td>
<td>x[1]</td>
<td>x[0]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>n = 2</td>
<td>x[2]</td>
<td>x[1]</td>
<td>x[0]</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Using a FIFO is not efficient because you are moving data around a lot – that costs time and energy.

A better method is to use a concern call pointer. This is shown in the diagram above. You fill an area of memory (or array) in sequence, and move the the pointer (shown as arrow) up a position as shown above.

Note that if you keep incrementing the pointer, it wraps around at the end back to the beginning in a circular way. This is known as a circular buffer.
Now let us consider how we write code in MicroPython to achieve all these. You will be doing it tomorrow in the Lab.

The interrupt service routine (ISR) is `isr_sampling`. We have to make `ptr` and `buffer_full` global variables because we need to access these OUTSIDE the routine. `s_buf` is also global – defined OUTSIDE the ISR.

The rest of the code is simple.
Now we will consider three methods in determining when a beat occurs. The simplest way is to calculate the instantaneous energy of the sound signal. We have done this before. Once you have calculated the instantaneous energy for every 20 ms 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 $\rho[n]$.
- Collect 1-2 second worth (i.e. 50 to 100 $\rho[n]$ values) and perform FFT on Matlab.
- The fundamental frequency of the spectrum $\rho[\omega]$ provides an estimate of the beat frequency.
- You could also do FFT on $x[n]$ itself. The frequency spectrum $X[n\Omega]$ of $x[n]$ gives us the colour of the sound signal.

In addition, you may also find the spectrum of $x[n]$. This gives you some information about the “colour” of the music. However, I found that it is rather difficult to deduce characteristic of music from the spectrum.
Second method is an improvement. Apart from computing the instantaneous energy, you can also compute the steady state energy by averaging 100 instantaneous energy readings. A beat is deemed to have occurred if determining the energy rises above a threshold.
The problem of the previous method is that if you choose the wrong value for $b$, the algorithm will not work well.

The threshold $b$ needs to adapt to the music itself. How?

Computer the variance $v[n]$ of the instantaneous energy $\rho[n]$ over a 20-msec window:

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

Now compute the threshold value $b$ as:

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

and try $\beta = 1.5$, and $\alpha = 0.0025$

Finally, you can even adapt the threshold value $b$ according to the music itself by computing the variance of the instantaneous energy. This is shown above.
Finally, you could include the frequency spectrum information in order to determine the beat. However, I don’t recommend you doing this on MicroPython - we do not have FFT routines written for the Pyboard, and it would take too long anyway.
For Milestone 3, you need to somehow generate a dance routine in order to synchronize to the beat of the music.

You should store the dance routine as a text file and transfer that to the SD card. For example, you could have a code such as:

F4B4R4L4....

For forward four beats, backward 4 beats, right turn 4 beats, left turn 4 beats ....
Beat synchronisation

- Although most of the signal processing work is done on Matlab, you still need to perform ONE real-time signal processing on Pybench using Python.
- You need to synchronous your dance move to live music, by analysing the instantaneous energy vs average energy, and move according to the beat.
Since motor coils are essentially inductors, they have low DC impedances (resistance of the wiring). Hence when driving motors, we need to use special driver chips.

The driver chip you used in Lab 4 (the TB6612) is often called the H-Bridge Driver. Shown here is the simplified block diagram. There are four transistors connected to the supply rail and ground. (It doesn’t matter which is which because the circuit is symmetrical.) The motor is connected in the middle forming the horizontal link of the H. The transistors are MOSFETs (metal oxide silicon field effect transistors) which is acting like a voltage controlled switch. When a ‘1’ or high voltage is applied to the gate control terminal, the transistor turns ON and conduct electricity. If a ‘0’ or low voltage is applied, the transistor is OFF. So the top diagram shows a configuration that results in the supply voltage being applied to the left terminal of the motor. The right terminal of the motor is grounded, and the motor turns in one direction. Reversing the control to the transistors results in the motor turning in the other direction.

If you use an AND gate at the control input, you can also add a PWM signal to control the speed of the motor.

Basically the ‘1’ and ‘0’ control signals are the A0 and A1 signals on the TB6612. The PWM signal is what you apply to the input of the AND gate.

Now you know how the TB6612 works.
Here is another package provided for you to implement Milestone 3. It relates to the microphone buffer, but packaged together as a python package. It is very easy to understand.