In this lecture, we will go through what you have learned in Lab 4. In particular, I will explain the limitation of polling as a method of detecting a real-time event. Then I will explain what and why interrupt is so much better.

Finally I will explain how you can use interrupt with the Pybench board using Micropython.

The content of this slide was presented to you last year in DE1.3 Electronics 1 module in Lecture 14 – Drive. I present this here again to refresh your memory.

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 will use 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 conducts 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.
Last year, you wired your motors to the TB6612 H-bridge driver chip (the one in brown colour) yourself. This year, the wirings are all done for you on the Pybench PCB.

You will also be connecting the Pybench board to a 12V lithium battery via the large yellow power connectors. The toggle switch turns this ON and OFF.

The motor is connected to the Pybench board via the 6-way ribbon cable. You will be learning how to drive the motors (again) in Lab 4 on Wednesday.

You will also be using the Bluetooth board (BluFruit UART from Adafruit) to control the speed of the motor. Here are the pin names on the Pyboard for reference.

Exercise 1 of Lab 4 is just a revision from last year’s Electronic 1 module. If you have forgotten this, please go back to last year’s lecture on “Drive”.

Here are some interesting questions to ask yourself to check whether you have learned what is expected of you:

1. Why do you need this driver chip at all? Could you drive the motor directly from the microprocessor?
2. How are the two pins (IN1 and IN2) used to control the direction of the motor?
3. What is PWM and why is it desirable to use PWM to control the speed of the motor instead of using analogue voltage level (e.g. from a DAC signal)?
4. What is meant by “Creating a pin object A1” in the Python code?
5. Explain how timer 2 is programmed to produce the PWM signal to drive motor in the following lines.

```python
# Configure timer 2 to produce 1kHz clock for PWM control
# Configure timer 2 to produce 1kHz clock for PWM control
tim = Timer(2, freq = 1000)
motor = tim.channel(1, Timer.PWM, pin = PWMA)
def A_forward(value):
    A1.low()
    A2.high()
    motorA.pulse_width_percent(value)
A_forward(50)
```

6. How should you choose the frequency of the PWM signal to drive the motor?
Next, we use the potentiometer (5kΩ) to control motor speed and direction. Here are the questions to test yourself:

1. In Micropython, how do you create an object to perform ADC conversion? Why in this case, we use pin X11?
2. How do you define and work out the resolution of the ADC converter?
3. Explain the meaning of the statement:

   \[
   \text{speed} = \text{int}((\text{pot.read()}-2048)\times200/4096)
   \]

4. Explain the meaning of the format statement in Python:

   \[
   \text{oled.draw_text(0,40,}'\text{Motor Drive:}\{5d\}\text{%.format(speed)}\text{)}
   \]

Next, we use the Hall Effect Sensors (two) on the motor to determine the speed of the motor and direction of the motor. The questions to ask yourself are:

1. Refer to the sensor output signals, what happens when you increase the motor speed?
2. How would the two sensor signals differ when you change the direction of rotation in the motor?
3. Given the waveform of the two signals (Channel A and B) from the sensors, the relative phase is always ±π/2. Why?
4. Given the circular magnet has 13 pole pairs, and that the gear of the motor has a 1:30 reduction ratio, how can you derive the speed of motor (in revolutions per second) from the number of rising edges E in a period T? (answer: 390 pulses per revolution. Therefore speed of motor is:

   \[
   \text{motor_speed (in rps) = (number of pulses/390) / T in seconds}
   \]
Pseudo code to measure speed by polling

- Initialize variables to zero: motor_speed, sensor_state, pulse_count
- Repeat forever:
  - Mark current time (as tic)
  - If sensor has gone from low to high (rising edge)
    - increment pulse_count
  - Update sensor_state by reading hall effect sensor value
  - If elapsed_time >= 100ms
    - motor_speed = pulse_count
    - reset pulse_count
  - display speed on OLED as motor_speed/39

Discuss: what is the limitation of polling?

This is typically how one can measure the motor speed by polling – continuously checking in a tight loop whether something has happened or not.

Here there are TWO polling operation happening. The first if-statement checks to see if the Hall Effect signal has a rising edge (goes from low to high). The second if-statement checks for a time window of 100msec. By counting the number of pulses detected in 100ms window, we can calculate the speed of the motor using the formula:

\[ \text{motor_speed (in rps)} = \frac{\text{number of pulses}}{39} \]

39 because each revolution of the motor generates 390 pulses. Therefore in a 100msec period, one revolution will give us 39 pulses!

Measure motor speed by polling

- Polling means checking for some event in a loop, then do something
- Here we check sensor signal of motor A changing from low to high in the polling loop
- When this occurs, increment a counter A_count
- We also check elapsed time = 100msec in polling loop (tic-toc)
- If time out, save count as speed measurement A_speed, and reset counter

We measure the speed of rotation by counting the number of low-to-high transitions on one of the two Hall Effect Sensor signals.
This can be achieved by polling – checking in the code when such transition has occurred. If yes, up a counter value. Then check if 100msec has elapsed. If yes, remember the count value and reset the counter.

Questions to ask yourself:
1. What is the purpose of these two lines?

   ```
   if (A_state == 0) and (A_sense.value() ==1): # rising edge detect
   A_count += 1
   A_state = A_sense.value() # read value on pin A_sense
   ```

2. How are tic and toc, which are built-in functions in Matlab, implemented in Micropython?

3. Explain the following codes:
Lab 4: The idea of interrupt

1. Save the state of program
2. Jump to ISR
3. Stop further interrupts
4. When finish return

- Hardware method to detect event (e.g. rising edge on a pin), generate interrupt
- Processor forced to do something else – defined in the Interrupt Service Routine (ISR)
- Return when finished

The reason why polling is not a good method to measure speed of motor is that microprocessor can only execute ONE instruction stream at a time. If you are checking (polling) for rising edge, you cannot do other things. Conversely if you are doing other things, you will miss the rising edges. That’s why in the experiment, you found that the polling method give a speed reading that is “noisy”, meaning that it is jumps all over the place!

Interrupt is different. You use HARDWARE method to detect the occurrence of an event. Forces the processor to suspend whatever it is doing at the time, and go to another segment of CODE to service the interrupt (hence we call this the Interrupt Service Routine or ISR).

When finished, return to the interrupted code and continue as before.

Question to ask yourself:
1. Why is interrupt better than polling?
2. What happens if your interrupt service routine is long and complex?
3. How should you think about a system with multiple interrupts?
4. What is it meant by “saving the state of the program”? Why is this necessary?

Lab 4: Interrupt Service Routines

- Need to detect and handle two types of events:
  1. Rising edge on Hall effect sensor signal on Y4
  2. 100ms elapsed time on a Timer
- Need two ISRs for these two interrupt events
- Need to provide a dummy variable as shown here

```python
def isr_motorA(dummy):
    # motor sensor ISR - just count transitions
    global A_count
    A_count += 1

def isr_speed_timer(dummy):
    # timer interrupt at 100msec intervals
    global A_count
    global A_speed
    A_speed = A_count
    A_count = 0
```

Here are two interrupt service routines. The first to handle low-to-high transition on the sensor signal from Motor A. The second to handle timer alarm which happens every 100msec.

Question to ask yourself:
1. When will the functions isr_MotorA and isr_speed_timer be executed?
2. What are the purposes of these two functions?
3. Why do you need to define A_count and A_speed as global?
Lab 4: setting up the interrupts

- Allocate some buffer space to handle errors
- Specify Pin Y4 as source of interrupt, rising edge
- Define timer 4 as a 100msec period timer (10Hz)
- timer.callback (ISR) - tell timer to generate an interrupt at end of period, and execute ISR

```python
# Create external interrupts for motorA Hall Effect sensor
import micropython
micropython.alloc_emergency_exception_buf(100)
from pyb import ExtInt

motorA_int = ExtInt ('Y4', ExtInt.IRQ_RISING, Pin.PULL_NONE, isr_motorA)

# Create timer interrupts at 100 msec intervals
speed_timer = pyb.Timer(4, freq=10)
speed_timer.callback(isr_speed_timer)
```

How does one set up interrupts in MicroPython using the Pyboard and the Pybench System? First you need to include the following statement to allocate memory to store the state of the program:

```
micropython.alloc_emergency_exception_buf(100)
```

Then you have to tell that hardware that pin Y4 will generate an interrupt on every rising edge, and that the interrupt service routine is `isr_motorA`:

```python
motorA_int = ExtInt ('Y4', ExtInt.IRQ_RISING, Pin.PULL_NONE, isr_motorA)
```

Then, you need to program Timer 4 to time out every 100msec:

```
speed_timer = pyb.Timer(4, freq=10)
```

Finally, you need to tell this Timer that it should generate an interrupt when time out, and run `isr_speed_timer`:

```
speed_timer.callback(isr_speed_timer)
```

Lab 4 – Interrupt MAGIC

- Program loop does not deal with motor sensor edge, not 100msec elapsed time
- A_speed will always contain instantaneous speed count

Once interrupt is set up properly, the main program loop only controls the motor. Measuring the speed of motor is done automatically. The global variable `A_speed` will contain the correct number of transitions in a 100msec window ALL THE TIME, and updated every 100msec automatically.
For your information and future reference, here are the pin assignment found on the Pybench board.

<table>
<thead>
<tr>
<th>PIN</th>
<th>FUNCTION</th>
<th>PIN</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>X1</td>
<td>Motor PWM_A/Servo 1</td>
<td>Y1</td>
<td>BLE-UART Tx</td>
</tr>
<tr>
<td>X2</td>
<td>Motor PWM_B/Servo 2</td>
<td>Y2</td>
<td>BLE-UAR Rx</td>
</tr>
<tr>
<td>X3</td>
<td>Motor control AIN1/Servo 3</td>
<td>Y3</td>
<td>SW1</td>
</tr>
<tr>
<td>X4</td>
<td>Motor control AIN2/Servo 4</td>
<td>Y4</td>
<td>Motor sensor A_A</td>
</tr>
<tr>
<td>X5</td>
<td>Analogue OUTPUT</td>
<td>Y5</td>
<td>Motor sensor A_B</td>
</tr>
<tr>
<td>X6</td>
<td>SW0</td>
<td>Y6</td>
<td>Motor sensor B_A</td>
</tr>
<tr>
<td>X7</td>
<td>Motor control BIN1</td>
<td>Y7</td>
<td>Motor sensor B_B</td>
</tr>
<tr>
<td>X8</td>
<td>Motor control BIN2</td>
<td>Y8</td>
<td>OLED-i2C RST</td>
</tr>
<tr>
<td>X9</td>
<td>IMU-i2C SCL</td>
<td>Y9</td>
<td>OLED-i2C SCL</td>
</tr>
<tr>
<td>X10</td>
<td>IMU-i2C SDA</td>
<td>Y10</td>
<td>OLED-i2C SDA</td>
</tr>
<tr>
<td>X11</td>
<td>POTS5</td>
<td>Y11</td>
<td>Microphone amplifier</td>
</tr>
<tr>
<td>X12</td>
<td>Analogue INPUT</td>
<td>Y12</td>
<td>Unused</td>
</tr>
</tbody>
</table>