Electronics Design Laboratory
Lecture #9
Lab 4 notes

- Finishing Lab 4 this week
- Demo requires position control using interrupts and two actions
  - **Rotate** a given angle
  - Move forward a given **distance**
- **Tuesday**: finish Lab 4 Part B
- **Thursday**: Lab 4 demo with a battery-pack powered robot
  - Stop, wait for the switch to be in the ON position
  - Wait 1 second
  - Move forward 2-feet
  - 180° clockwise rotation of the robot
  - Move forward 2-feet
  - 180° counter clockwise rotation of the robot

**Accuracy**: the robot should come back to the starting position
Schedule updates: see Lab calendar

• Lab 4 report due by 5pm on Sunday April 8
• Project presentation slides due by 11:30pm on Monday, April 9
• Project proposal presentations will in the Lab on Tuesday, April 10
Need a relation between our control variables ($V_{\text{ref}}$ and On/Off) and our output variable (position). Two approaches are frequently used.

• **Speed Control**
  – Relies on the fact the distance = speed * time
  – Uses no external inputs, simple to implement, and inaccurate

• **Position Control**
  – Directly measures motor position to set robot position
  – Needs external inputs, more complicated, but more accurate
Position Control based on Speed Control (Theory)

In steady state ... \( \omega = \frac{V_{\text{ref}}}{K_{\text{sense}}} \)

Distance:
\[
\text{distance} = \left( \frac{\omega}{2\pi} \right) (\text{time}) = \left( \frac{rV_{\text{ref}}}{K_{\text{sense}}} \right) (\text{time})
\]

Rotation:
\[
\text{rotation} = \left( \frac{360}{2\pi} \right) \left( \frac{\text{distance}}{r_w} \right) = \left( \frac{180rV_{\text{ref}}}{\pi r_w K_{\text{sense}}} \right) (\text{time})
\]

The robot wheel radius is \( r = 6.5 \text{ cm} \)
Position Control based on Speed Control ($\mu$C code example)

```c
// define pins
#define PINON  6  // connect pin 6 to ON/OFF switch, active HIGH
#define PWMLEFT  9  // connect pin 9 to speed reference (left)
#define PWMRIGHT 10 // connect pin 9 to speed reference (left)

// setup pins and initial values
void setup() {
    pinMode(PINON, INPUT); // define PINON as an input
    pinMode(PWMLEFT, OUTPUT); // define PWMLEFT as an output
    analogWrite(PWMLEFT, 0); // set speed reference
}

void loop() {
    while(digitalRead(PINON) == LOW){}
    delay(2000); // wait 2 second
    analogWrite(PWMLEFT, 128); // set speed reference, duty-cycle = 128/255
    analogWrite(PWMRIGHT, 128); // set speed reference, duty-cycle = 128/255
    delay(500);
    analogWrite(PWMLEFT, 0); // set speed reference
    analogWrite(PWMRIGHT, 0); // set speed reference
}
```

- **Example:** $K_{sense} = 0.46$, want to go forward 18cm.
  - Decide that we want to go forward at half speed (assuming here full speed is achieved at $V_{ref} = 5V$)
  - Calculate required travel time using eqs on the previous page = 500ms
Practical issues with the distance = speed*time approach

\[
\text{distance} = \left( \frac{r V_{\text{ref}}}{K_{\text{sense}}} \right) (\text{time})
\]

- \(K_{\text{sense}}\) will vary between wheels: \(K_{\text{sense}} \approx 610 t_{\text{on}} \approx 610(R_2C_2 \ln(3))\)
- \(R_2\) and \(C_2\) have tolerances, so \(K_{\text{sense}}\) may be off: try to tune \(R_2\) or \(C_2\) to match the speed sensor \(t_{\text{on}}\) between the two wheels!
- Speed has dynamic responses, it is not always equal to the steady-state value; one wheel may speed up faster or slower than the other

---

Vref = 2.5V for 500ms

Red Wheel Starts 20% faster than Blue Wheel

Neither wheel reaches distance calculated

Blue wheel ends behind red wheel
Better: position control based on counting encoder pulses

**encoder**

\[
\begin{pmatrix}
\text{pulses} & & \text{pulses}
\end{pmatrix}
\begin{pmatrix}
\text{rotation} & \text{rotations} & \text{turns}
\end{pmatrix}
\begin{pmatrix}
\text{rotation} & \text{turn} & \text{distance}
\end{pmatrix}
\begin{pmatrix}
\text{pulses} & & \text{pulses}
\end{pmatrix}
\]

Input to output relation!

\[
(13)(51) \left( \frac{1}{2\pi r} \right) = \left( \frac{N_{\text{enc}}}{\text{distance}} \right)
\]

Distance \approx 0.62 \frac{\text{mm}}{\text{pulse}}
Encoder based Position Control

Each encoder pulse represents a fraction of a wheel turn
Distance read directly, without guessing speed/distance relation

This is better... but how do we read encoder pulses?
- The simple approach is called 'polling', or 'busy-wait'
- The better approach uses 'event-driven' programming using an interrupt
Position Control: Polling

```c
int target; // desired number of rotations
int enc_count_Left = 0; // reset count to zero
int encValue; // current encoder pin value
/* probably more defs here */

void setup(){ /* all the usual stuff*/ }

void loop(){
  /* some other stuff. pause, setup, etc */
  target = 3*768; // three rotations
  enc_count_Left = 0; // reset encoder count
digitalWrite(pinCW_Left,HIGH); // go clockwise
while(enc_count_Left < target) { // loop until target is reached
do {
    encValue = digitalRead(pinEncoder); // poll encoder output
} while (encValue == LOW); // until it goes HIGH
enc_count_Left++; // increment count
do {
    encValue = digitalRead(pinEncoder); // poll encoder output
} while (encValue == HIGH); // until it goes LOW
} /* we are in the right position! do some other stuff.*/
}
Number of pulses: coding options

```c
int target;       // desired number of rotations
int enc_count_Left = 0;   // reset count to zero
int encValue;        // current encoder pin value

/* probably not used? */

void setup()
{
    /* use an integer */
    const int const_RP = 663
    /* use a constant */
    #define CONST_RP 663
    /* use a macro */
    #define T2N(x) { x*663 }
    /* use a function */
    int t2n(rotations) { return rotations*663; }
    /* use a combination might be best */

    while (enc_count_Left < target) {
        do {
            encValue = digitalRead(pinEncoder);   // poll encoder output
        } while (encValue == LOW);          // until it goes HIGH
        enc_count_Left++;
        do {
            encValue = digitalRead(pinEncoder);   // poll encoder output
        } while (encValue == HIGH);          // until it goes LOW
    } /* we are in the right position! do some other stuff */
}
```
Position Control – Polling vs. Event Driven

Polling / Busy-Wait

- Wastes μC time
- Doesn’t allow other tasks to execute!

μC time

... μC waiting for transition...

Increment

- μC waiting for transition...

Increment

- μC waiting for transition...

Event Driven

- Allows other tasks to execute!

μC time

... μC executes other tasks...

Increment

... μC executes other tasks...

Increment

... μC executes other tasks...

Increment

- Want our program to do useful stuff between encoder pulses
- Ideally, when a rising encoder pulse is seen, our microcontroller switches tasks to a small, fast function to increment our encoder counter
- The small function is called an ‘interrupt service routine’, and handling inputs this way is called ‘event-driven input/output’ programming
Position Control – Event Driven

Event Driven
Allows other tasks to execute!

Encoder Output

µC time

ISR_count()

loop()

Loop code runs between encoder pulses

When a rising edge is detected, we quickly run ISR_count

When needed, loop can read the value of enc_count

No waiting required! Event-driven input/output is better in this case

volatile int enc_count = 0;

void setup(){
    attachInterrupt(0, ISR_count, RISING);
    Serial.begin(9600);
}

void loop(){
    delay(1000);
    /*do stuff!*/
    delay(1000);
    /*more stuff!*/
    Serial.write(enc_count);
}

void ISR_count(){
    enc_count++;
}
attachInterrupt pins on Arduino Uno: 0 = pin2, 1 = pin3

- Inputs from robot: **On/Off** (for future use) and **Encoder Pulses** (for future use)
- Outputs to robot: **Stop/Go controls** (2 per wheel) **Speed reference** (1 or 2 total)
- Voltages: **5VDC** for speed sensing circuits and **10VDC** for Motors/Encoders

**Interrupt 0** (pin 2):  
ENCODER_L (2)  
ENCODER_R (3)  
ON/OFF (6)  
CW_L (7)  
CC_L (8)  
REF_L (9)  
REF_R (10)  
CW_R (11)  
CC_R (12)  
LED (13)

**Interrupt 1:**  
ENCODER_L (2)  
ENCODER_R (3)  
ON/OFF (6)  
CW_L (7)  
CC_L (8)  
REF_L (9)  
REF_R (10)  
CW_R (11)  
CC_R (12)  
LED (13)

Input from bench supply or battery
- 10V
- GND
- 5V

Output used to supply +5V circuitry (Labs 2 and 3)
Finer positioning details: Setting Speed Reference

**Inputs**
- Desired Position

**Microcontroller**
- Target $N_{enc}$
- Control Code

**Motor System**
- $V_{ref}$
- $\omega$

**Simple approach leads to overshoot...**

Desired Position

Actual Position

Robot Speed

Momentum keeps robot moving

Start Action

Encoder Count Reached

Robot Stops Moving

time
Finer positioning details: setting speed reference as a function of distance traveled

- Lowering $V_{\text{ref}}$ can reduce but not remove overshoot.
- Either:
  - Offset the target $N_{\text{enc}}$ because you know you will overshoot, or
  - **Be smarter about setting $V_{\text{ref}}$**

Example:

$$
V_{\text{ref}}^{\text{start}} = \text{Min}\left[ V_{\text{ref}}^{\text{Max}}, \beta(N_{\text{actual}}) \right]
$$

$$
V_{\text{ref}}^{\text{end}} = \text{Min}\left[ V_{\text{ref}}^{\text{Max}}, \beta(N_{\text{target}} - N_{\text{enc}}) \right]
$$

$$
V_{\text{ref}} = \text{Min}\left[ V_{\text{ref}}^{\text{start}}, V_{\text{ref}}^{\text{end}} \right]
$$

![Diagram showing robot speed and actual position over time, with V_max and actual position indicated.](image)
Lab 4 Demo

• Show how the robot powered from 2 battery packs in series (approximately 10 V) can accomplish the specified Part B.2 positioning task:
  – Stop, wait for the switch to be in the ON position
  – Wait 1 second
  – Move forward 2-feet
  – 180° clockwise rotation of the robot
  – Move forward 2-feet
  – 180° counter-clockwise rotation of the robot

**Accuracy: the robot should come back to the starting position**

• Show your position control program
• Show complete speed control circuit, and complete LTspice diagram of your speed control circuit
• Answer questions related to your position control code and speed control circuit

This Lab includes an extra credit opportunity: see next page
Lab 4, B.4 Extra Credit

The groups whose robot accurately completes the following tasks will be eligible for extra credit:

- Robots will start centered on a floor intersection facing ‘north’
- From the starting position, robot must perform the following moves, as shown on the diagram:
  - Move forward one square ‘north’
  - Turn 90° CC
  - Move forward two squares ‘west’
  - Turn 90° CW
  - Move forward one square ‘north’
  - Turn 90° CW
  - Move forward one square ‘east’
  - Turn 90° CW
  - Move forward two squares ‘south’
  - Turn 90° CC
  - Move forward one square ‘east’
  - Turn 90° CC and stop
- At the end, the robot should ideally be in the starting position.

Accuracy requirements:
- Robot platform must always cover a portion of the floor line along the route shown in the diagram
- In the end position, both wheel axis tips must be within +/- 5cm (+/- 2”) of the southern horizontal floor line, and the caster wheel must be within +/- 5cm (+/- 2”) of the eastern vertical floor line
Appendix

Some basic µC topics: word length, interrupts, serial communication
What is the ‘word length’ of a processor?

- This is the bit-size that the processors instruction set operates on, and generally the size of the processors data bus and ALUs
- This is sometimes referred to as the ‘natural’ unit that the processor uses
- The Arduino UNO uses an 8-bit processor,
  - with an 8-bit instruction word size
  - with an 8-bit data word size (data bus is 8-bits)
  - with a 16-bit address word length...
- Not an exact term. Used in many ways. Confusing, and needs a modifier to make any sense
- Old processors (pre 1965) used 6-bit word length. First 8-bit mainframe was the ‘System/360’

How does serial work?

- Many standards (I²C, RS-232, ‘something you made up’, etc..)
- Lets go over simplified RS-232
"An **interrupt** is a signal to the processor **emitted by hardware or software** indicating an event that needs attention."

Interrupt vectors are stored at the beginning of memory. Each vector stores a jump instruction and an address.

- Vector 0x000 jumps to 0x0032
- Vector 0x002 jumps to 0x0A00

The boot loader runs, then jumps into the main program.

Any code that you have written is stored in the middle of memory. Functions like loop(), setup(), etc. are stored here.

Arduino functions such as delay(), digitalRead(), etc. are all stored here, after your code. There may also be some free space, which the compiler might fill with zeros.

Program Memory (FLASH)

```
0x0000  0x0032  0x0232  0xA00  0xFFFF

Interrupt Vectors
Boot Loader
loop()
ssetup()
isr_0()
Other Stuff
```

Program counter
Which memory address is my next instruction at?

Core
1. Execute instruction
2. Increment P. Counter
3. Fetch Next Instruction

Instruction Register
My new instruction!

Interrupt timeline
1. Core senses an interrupt!
2. Core checks if that interrupt is enabled
   a) If not enabled... ignore
3. Current PC is stored (PUSH)
4. PC is set to Correct Interrupt Vector
5. Core executes jump to ISR
6. ISR is executed as normal
7. At the end of the ISR, a special jump instruction ‘RETI’ is executed.
8. RETI resets PC to stored value (POP)
9. Core continues incrementing PC and executing instructions.

C:\~\seltzer> arv-obgdump -S prog.elf > temp.txt
- Two wires, a transmit (Tx) and a receive (Rx)
- Each wire has two states, High and Low
- A device will read on its receive line (digital in), and transmit on its transmit line (digital out)
- Say device A wants to send the character ‘a’ to device B
Serial Communication

- In ASCII, the character ‘a’ is 0x61 = 0b01100001
- We can send this one bit at a time
  - A ‘time series’ of bits. One bit every time step
- We need to let device B know we are sending a character, so we add a start bit, 0b1. We also need to agree on how fast things will be sent. Let’s choose 1bit/ms
- We also need to let device B know when we are done, so we add a stop bit, 0b0
- Our final series of bits is ‘1011000010

```
const int Tx = 1;
int index;
int to_send[8] = {0,1,1,0,0,0,0,1};
digitalWrite(Tx,HIGH);
for(index = 0; index < 8; index++) {
  delay(1);
  if(to_send[index] == 0) digitalWrite(Tx,LOW);
  else digitalWrite(Tx,HIGH);
}
delay(1);
digitalWrite(Tx,LOW);
```

Device A

Device B

- GND

ECEN 2830  Electronics Design Laboratory  23
Serial Communication

- Device B now needs to read the value off of the line
- This is easiest to do with a shift register (Hardware implementation)
  - Shift registers can convert serial data (time series) to parallel data

We can implement this in code as well

```c
const int Rx = 0;
int index;
int received[8] = {0,0,0,0,0,0,0,0};
char answer;

while(digitalRead(Rx) != 1){}
for(index = 0; index < 8; index++) {
    delay(1);
    received[index] = digitalWrite(Rx);
}
answer = array2ascii(received);
```