Exam 2 Next Wednesday - 4/22

Not technically cumulative - material from Exam 1 built upon for this newer material, so tangentially needed.

Topics:

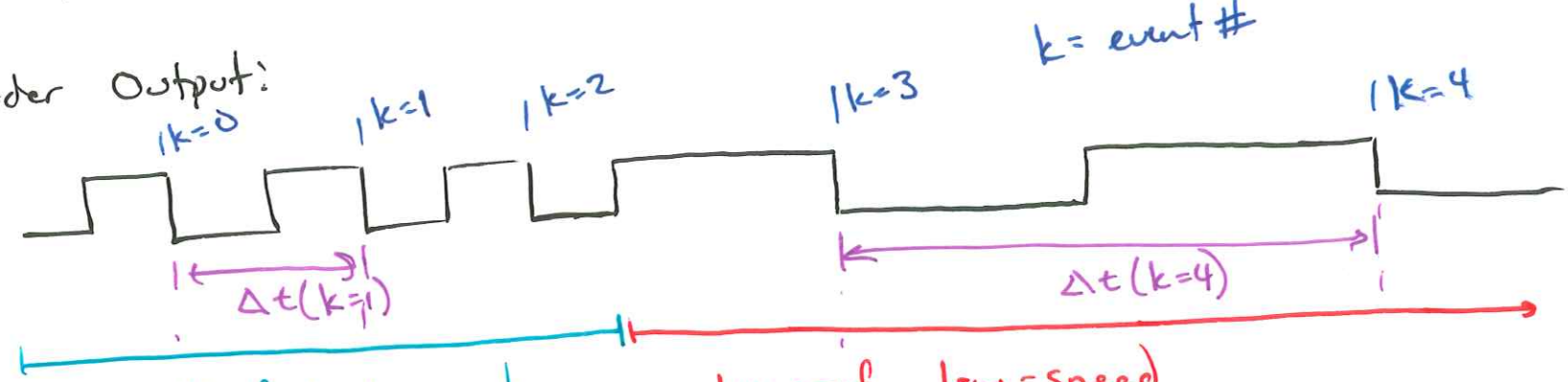
- Timers
  - Capture - Encoders, etc.
  - Compare - PWM, etc.
- ADC, Potentiometers.
- System Responses
- PID control
- I^2C

→ Short Review for Each.

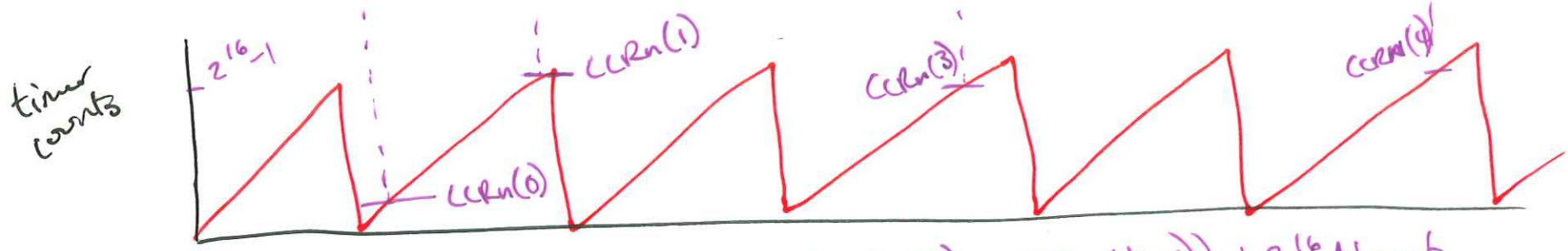
Timer Capture – Encoders

Purpose: Measure timing of events OR receive time-encoded information.

Ex Encoder Output:



CCRn(k) – value of timer at event k.

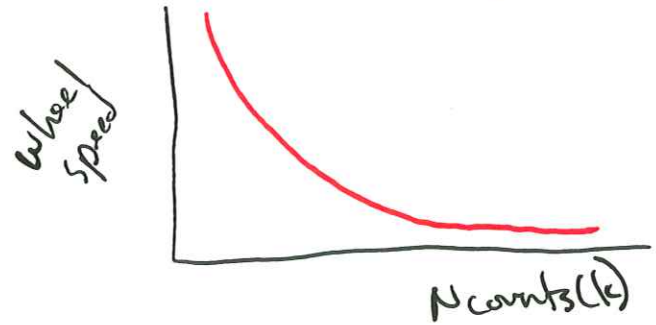


$N_{counts}(k)$ = counts between events = $(CCRn(k) - CCRn(k-1)) + 2^{16} N_{resets}$

$\Delta t(k) = N_{counts}(k) \cdot T_{inc}$

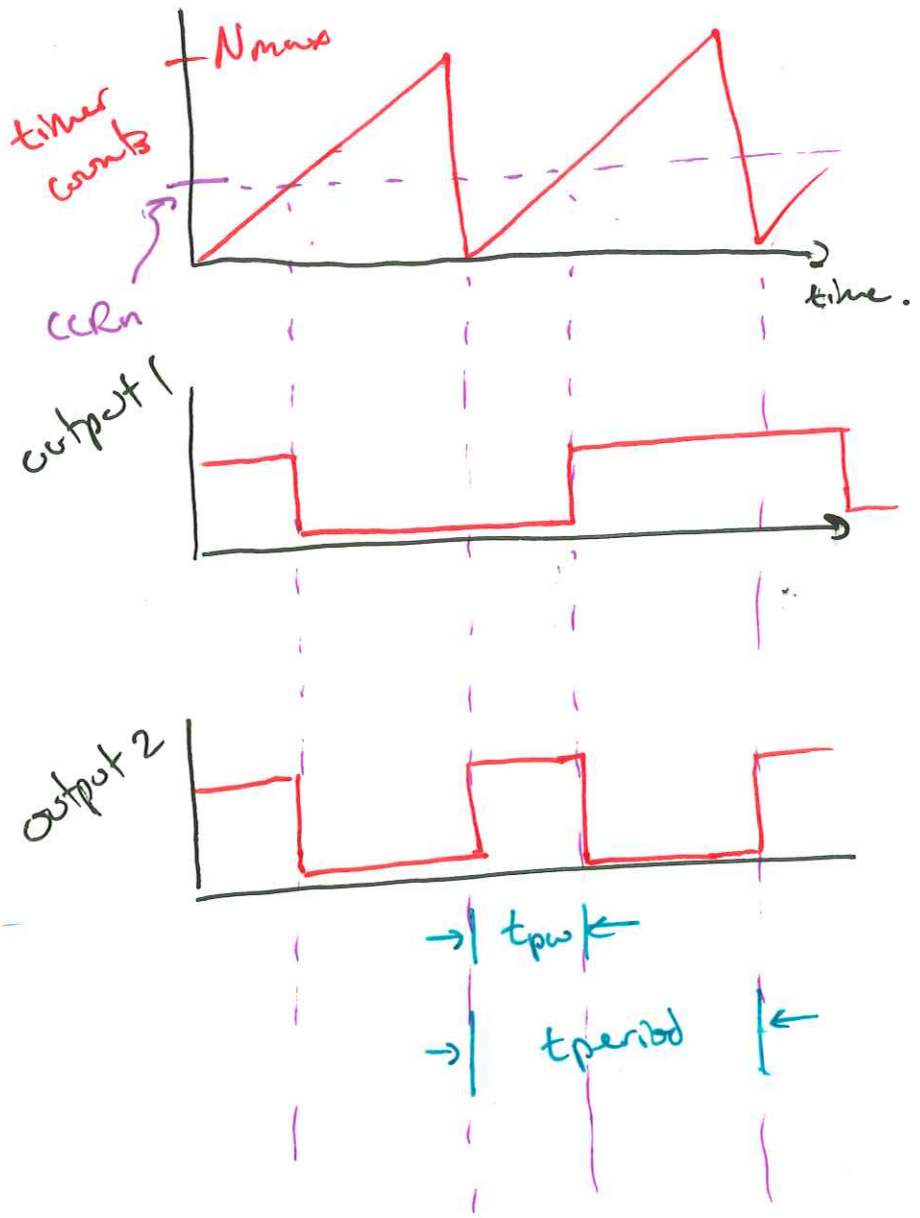
↑ time for each counter increment.

enc-counts from lab.



Timer Compare - PWM

CCRn - A comparison value set by code.



← toggle on ~~compare~~ compare.
Do nothing on reset.
(produces square wave)

← ~~set~~ clear on compare
set on timer reset
PWM output

understand implications of different Compare & timer reset configurations.

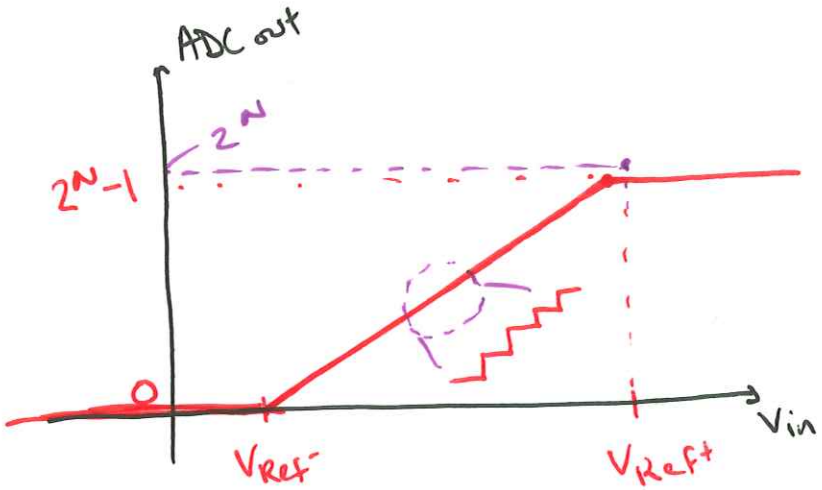
$$DC[\%] = \frac{t_{pw}}{t_{period}} \times 100\%$$

Adjust t_{pw} by changing CCRn

t_{period} usually fixed at a constant.

multiple CCRns can operate within 1 timer.

Analog-to-Digital Conversion, ADC, Read a raw voltage (or current) as a digital value. (N-bits)



$$\text{ADC output} = \begin{cases} 0 & V_{in} \le V_{Ref-} \\ \left\lfloor 2^N \frac{V_{in} - V_{Ref-}}{V_{Ref+} - V_{Ref-}} \right\rfloor & 0 < V_{in} < V_{Ref+} \\ 2^N - 1 & V_{in} \ge V_{Ref+} \end{cases}$$

$\lfloor x \rfloor$ → round x down.

V_{Ref-} ↓ usually 0

V_{Ref+} ↑ 1.4(3), 2.5V, 3.3V (internal) or other (external)

N in this case – Number of bits in conversion.

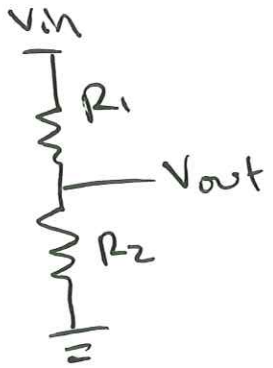
Estimate V_{in} value from adc output! (assume $V_{Ref-} = 0$)

$$\text{ADC output} = \left\lfloor 2^N \frac{V_{in}}{V_{Ref+}} \right\rfloor \rightarrow \hat{V}_{in} = V_{Ref+} \frac{\text{ADC output}}{2^N}$$

↑ inherently erroneous. (Quantization)

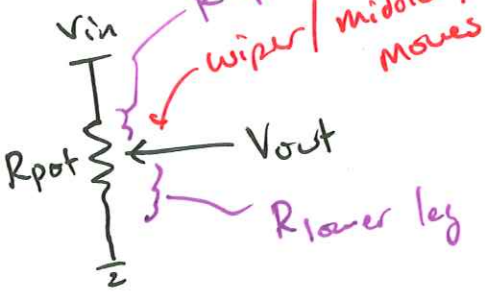
Maximum error expected $\hat{V}_{in}\Big|_{\text{ADC output} = \blacksquare 1}$

Potentiometers:



$$V_{out} = V_{in} \frac{R_2}{R_1 + R_2}$$

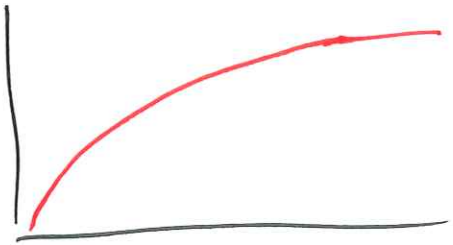
effectively this



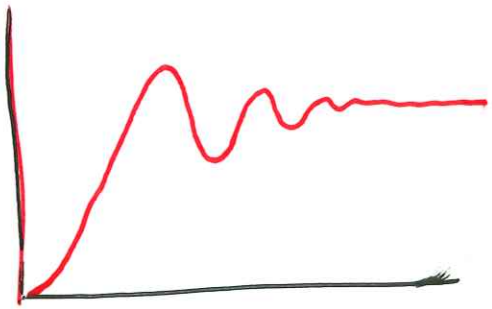
$$V_{out} = V_{in} \frac{R_{lower\ leg}}{R_{pot}}$$

System Responses

- Know Difference between 1st order & 2nd order (& 0th order)
- 1st Order:
  - Rise time
  - Settling time
  - Steady-State value / error
  - Time constant (τ)



- 2nd Order
  - All of above except τ
  - % overshoot
  - Oscillation frequency.
  - Damping types (~~over~~ overdamped, underdamped, crit. damped, unstable)



PID Control

P: Proportional → Acts on instantaneous error

I: Integral → Acts on error history

D: Derivative → Acts on rate-of-change of error

P: Most impactful when very far away from target / desired value.

larger k_P → faster rise time but more sensitive to noise.
may also make oscillation worse (or cause instability)

I: Most impactful when errors persist. (steady-state errors)

larger k_I → Reduce steady-state errors faster but
may make transient response (beginning of resp.)
worse / much worse.

D: most impactful when system "Moving fast", tries to stop "movement"

larger k_D → Increase system damping, reduce or remove
oscillations, but slow system resp.

PID equations

$e(t) = \text{desired}(t) - \text{actual}(t)$

Continuous Domain:

$$y(t) = k_p e(t) + k_I \int_0^t e(\tau)\, d\tau + k_D \frac{de(t)}{dt}$$

Discretized:

$$y(k) = k_p e(k) + k_I \sum_{n=0}^{k} e(k) + k_D \left(e(k) - e(k-1)\right)$$

$y(t), y(k)$ is control output value.

I²C : See last lecture for functionality

Example ① have device with addr = 0x66

config. device with Reg1 = 0x16, Reg2 = 0x53

1. Save values into an array.

```
uint8_t dat[] = {0x16, 0x53};
```

or

```
uint8_t dat[2];
dat[0] = 0x16;
dat[1] = 0x53;
```

2. Write values to device:

```
I2C_writeData([mod], 0x66, 1, dat, 2);
```

(address, start register, data to send IN ARRAY, # of bytes to send)

I2C Packet:

START (0x66<<1)+0 ACK 0x01 ACK 0x16 ACK 0x53 ACK STOP

Example ② Read 3 bytes from Device 0x2A, Registers 4 & 5, 7.

Reg 4 is low byte } of uint16-t
Reg 5 is high byte }

1. Create array

```
uint8_t dat[4];
```

2. Read the data:

```
I2C_readData(mod, 0x2A, 4, dat, 4)
```

→ gives: dat[0] = Reg. 4
[1] = Reg. 5
[2] = Reg. 6
[3] = Reg. 7

3. Assemble data:

```
uint16_t rg45 = (dat[1]<<8) + dat[0];
uint8_t rg7 = dat[3];
```

transactions

START (0x2A<<1)+0 ACK 0x04 ACK STOP
START (0x2A<<1)+1 ACK 0x16 ACK 0x2F ACK
0x77 ACK 0x01 NACK STOP