University of Nottingham
UK | CHINA | MALAYSIA

Lecture 8

Stepper Motors and Algorithms

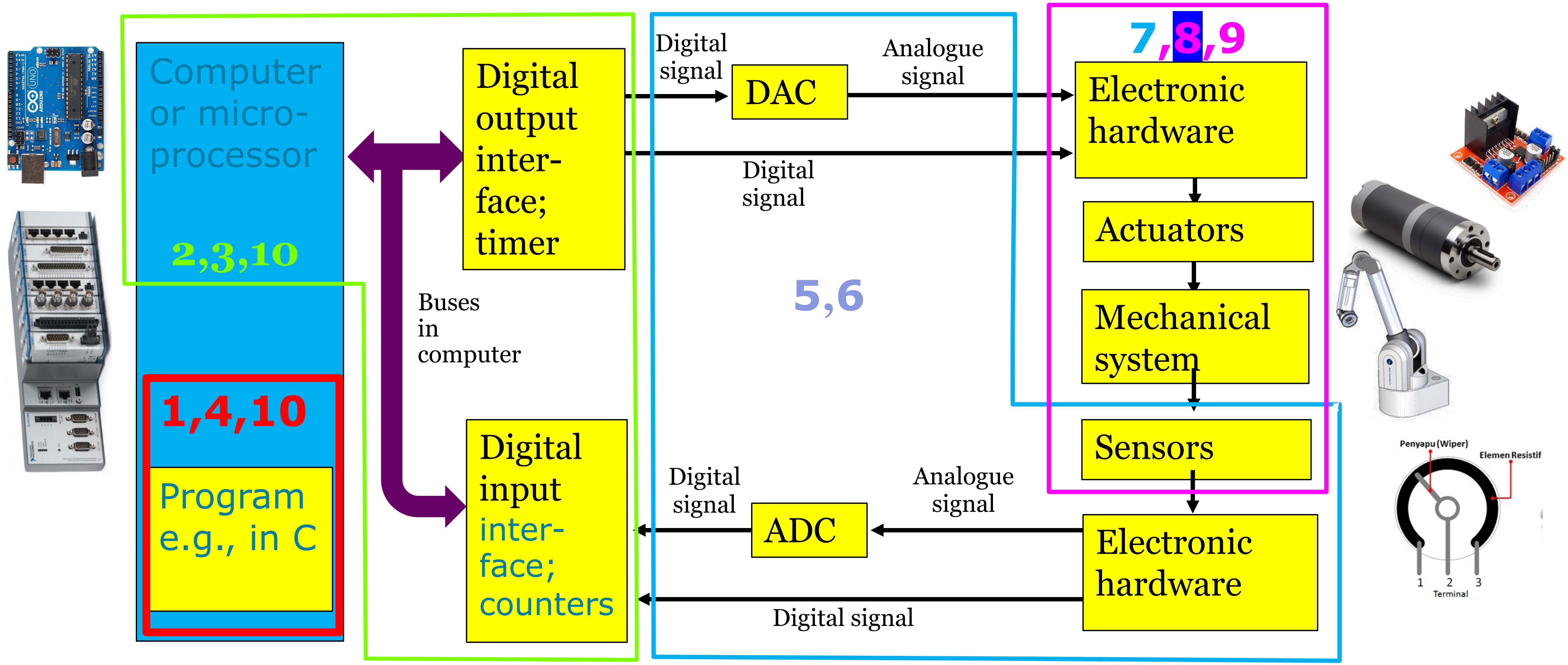
Mechatronics
MMME3085

Module Convenor – Abdelkhalick Mohammad

Lecture Objectives

- To understand how to *interface* a stepper motor to a computer
- To appreciate the issues associated with *generating the movements* for a stepper motor
- Understand the stepper motor *characteristics*
- To link the contents of this lecture and previous lectures on Motors to what you will see in *Lab 2*

A typical Mechatronics System



Recap

From last lectures

So far, we learned …

- How to deal with digital signals including train of pulses
  - Generate digital signal
  - Read digital signal
- Timer/Counters as a hardware solution
- Registers in µp
- State Tables
- Finite State Machines
- Interrupt
- DAC and ADC
- DC servo Motor & Stepper Motors

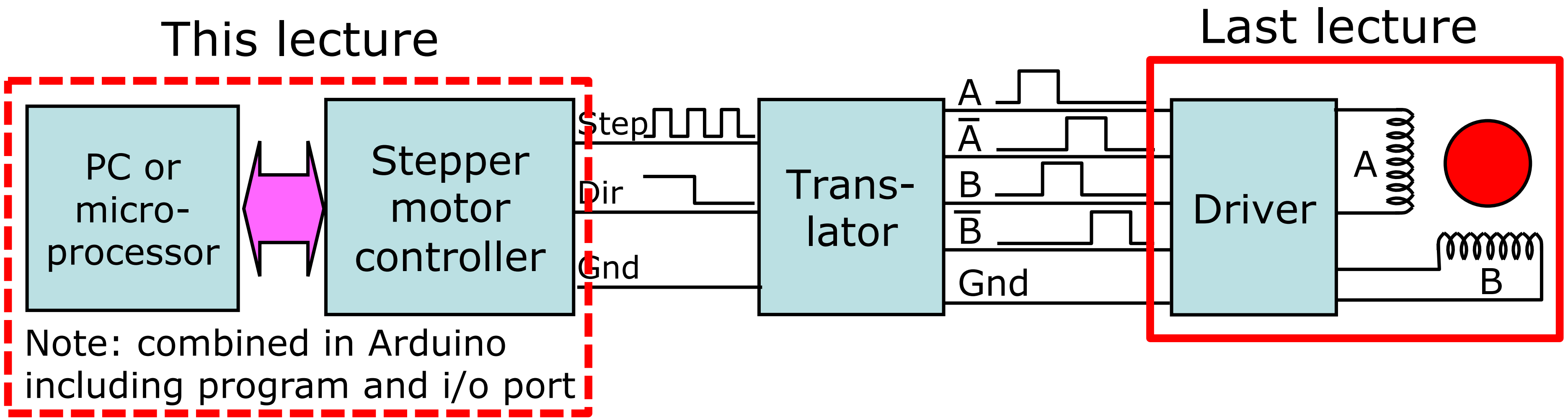
Stepper Motors Control

Introduction

Brief Revision

- Simple and convenient way of providing precise movement
- Normally open loop mode, no feedback
- They are used in a wide variety of applications including:
  - 3D printers and hobby CNC machines
  - Computer peripherals
  - Laboratory equipment
  - Student projects

What controls the stepper motor?



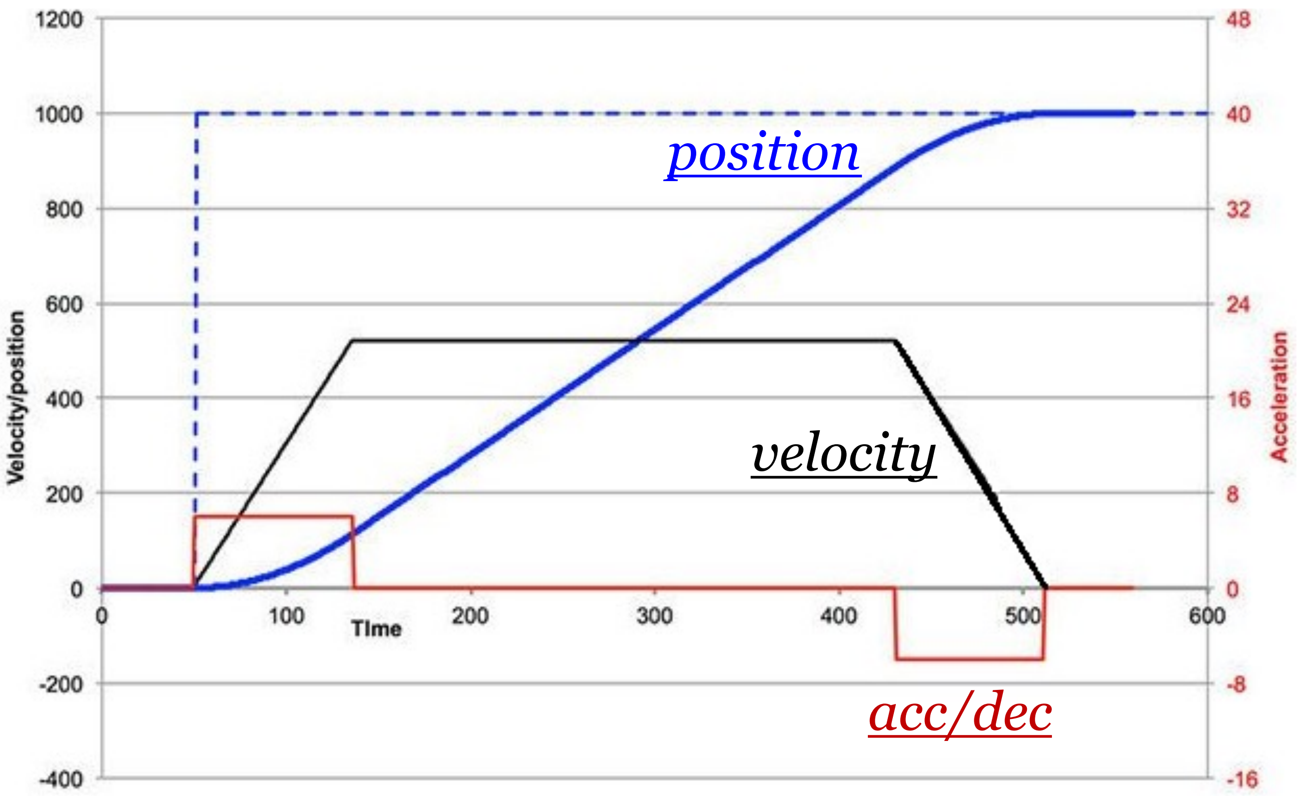
- We normally want to control stepper motor from a computer
- Need "step" and "direction" pulses to give required:
  - Number of steps
  - Maximum velocity
  - Acceleration/deceleration profile

Stepper Motors Control

Conversion of velocity profile to a steps

What is a motion profile?

Stepper Motors applications require defined, controlled movements, often to move a part to a specified position at a precise velocity or along a predetermined path. A *motion profile* provides the physical motion information and graphically depicts how the motor should behave during the movement (often in terms of *position*, *velocity*, and *acceleration*) and is used by the stepper controller to determine what commands (*steps*) to send to the motor.

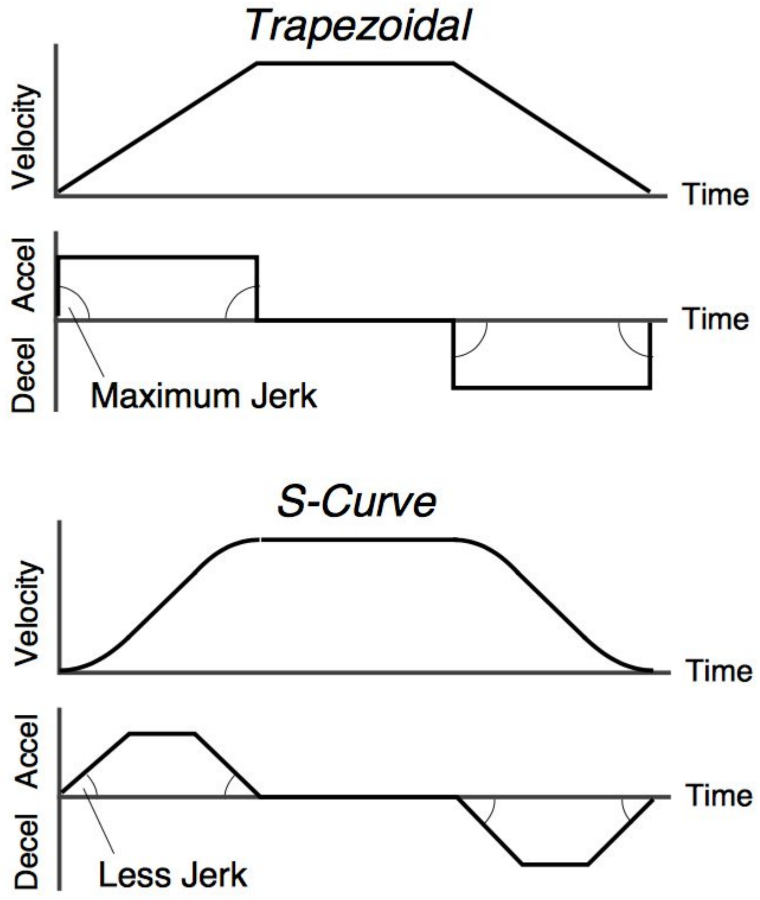


What is a velocity profile?!

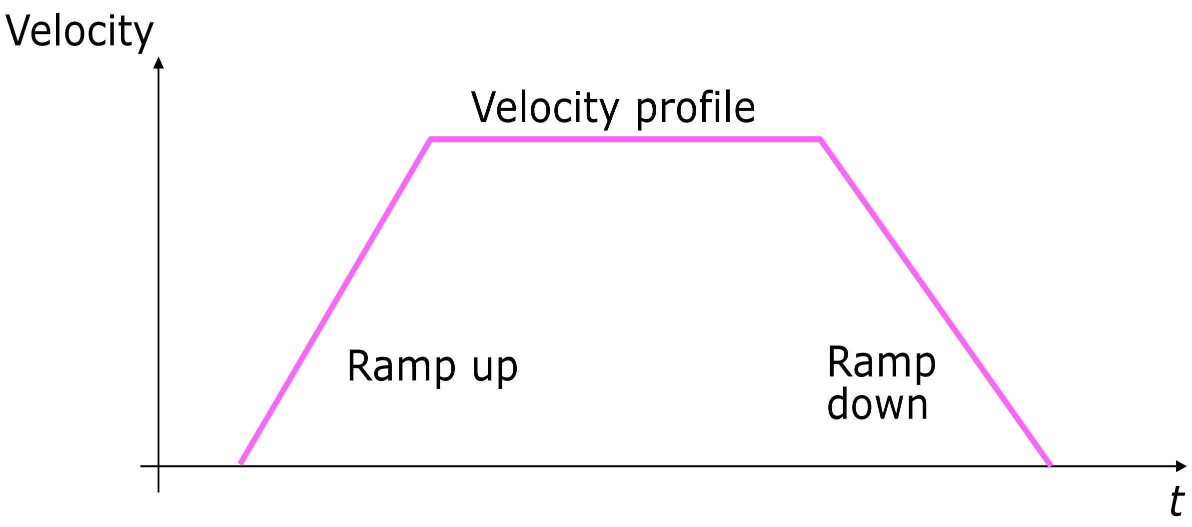
A velocity profile is a graph that shows how the speed of a stepper motor changes over time. It is used to control the motor's acceleration, deceleration, and maximum speed. There are several different types of velocity profiles, but the most common are:

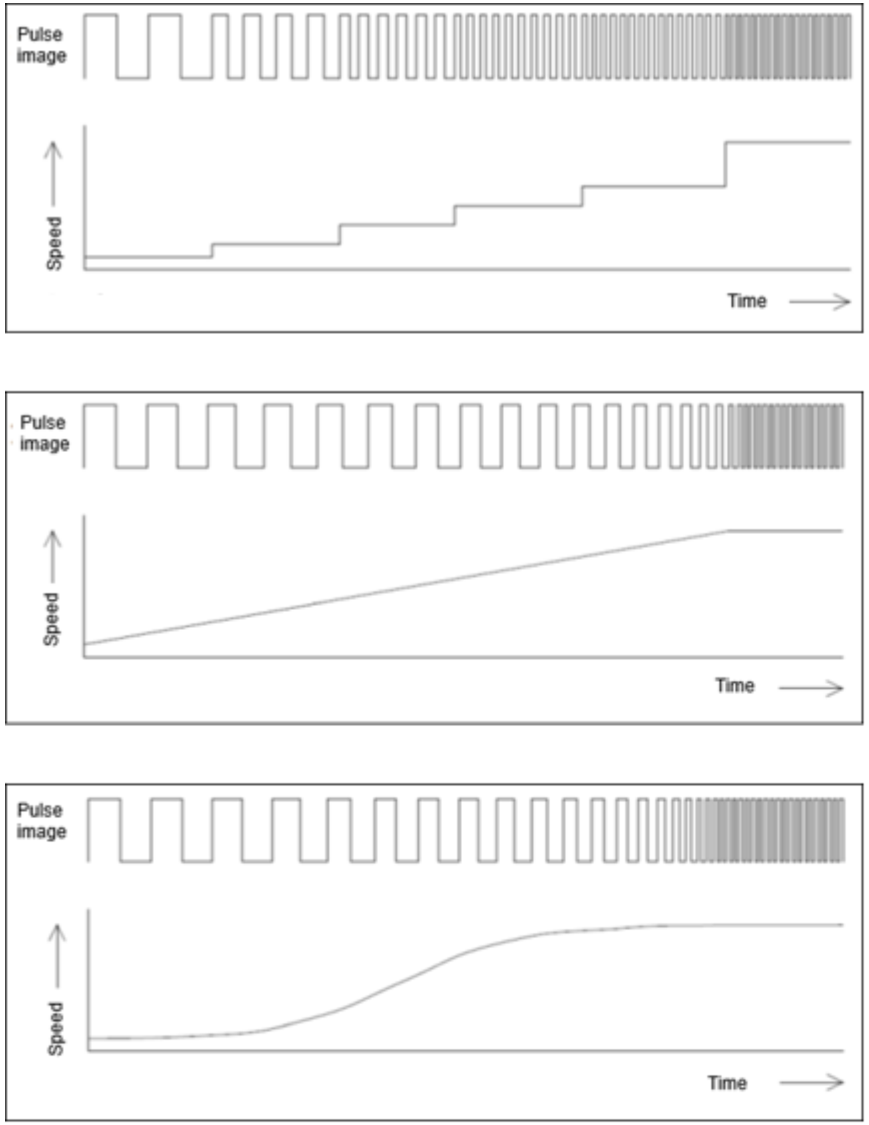
- **Trapezoidal profile**: This is the simplest type of profile and is easy to implement. It consists of three phases: acceleration, constant speed, and deceleration.

- **S-curve profile**: This is the most complex type of profile and is used for applications that require very smooth motion.



How to convert the Velocity profiles → Steps?!





Time per Step Vs Step per Time

There are two main approaches to step generation [1]:

Time per step:

- Calculate time until next step is due
- Keep checking if time has elapsed
- If time elapsed, make step
- Re-calculate interval then repeat as above

- This approach is used in Leib Ramp algorithm, AccelStepper library, GRBL
- We will focus on **time per step** algorithms and how they are implemented

Step per time:

- Calculate time since last step
- Multiply it by current speed to get desired distance moved
- Keep doing the above until it exceeds 1 step
- Make step and re-calculate speed, repeat.

[1] M Y Stoychitch 2018 *IOP Conf. Ser.: Mater. Sci. Eng.* **294** 012055

Stepper Motors Control

How can we generate the steps on a hardware?!

Time per step: Implementation 1

- Lab 2 experiment illustrates two approaches to implementing “time per step”:
  - A **timed loop**, like in “BlinkWithoutDelay” (but in μs), here *p* is time per step:

```
/* Timed loop for stepping, and associated coding */
currentMicros = micros();
if (currentMicros - prevStepTime >= p)
{
  moveOneStep();
  prevStepTime = currentMicros;
  computeNewSpeed();
}
```

Timed loop code

Actually, make a step pulse

p is re-calculated here for next step

Time per step: Implementation 1

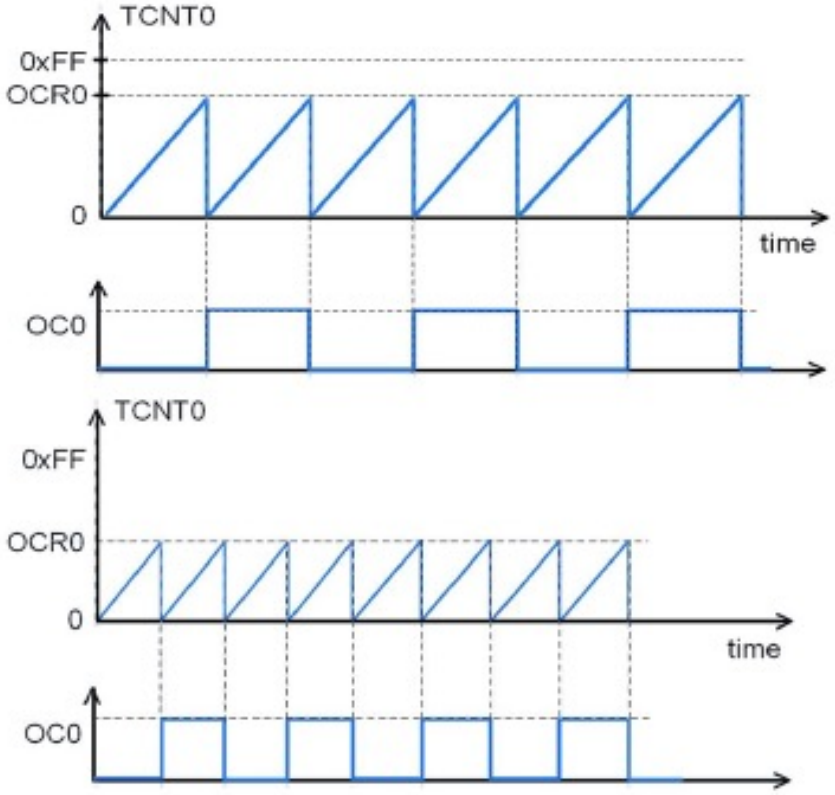
```
void moveOneStep()
/* Move a single step, holding pulse high -
{
  if (p != 0) /* p=0 MEANS "don't step" */
  {
    digitalWrite(stepPin, HIGH);
    if (direction == FWDS)
    {
      digitalWrite(dirPin, HIGH);
      currentPosition++;
    }
    else
    {
      digitalWrite(dirPin, LOW);
      currentPosition--;
    }
    delayMicroseconds(stepLengthMus);
    digitalWrite(stepPin, LOW);
  }
}
```

Time per step: Implementation 2

Remember this from Lecture 3 ☺!

2. Clear Timer on Compare Match (CTC) Mode

- In CTC mode the counter *is cleared to zero* when the counter value (TCNTn) matches either the OCRnA or the ICRn (*later we see this*).
- The OCRnA or ICRn define the top value for the counter, hence also its resolution.
- This mode allows greater control of the compare match output frequency.
- It also simplifies the operation of counting external events.



Time per step: Implementation 2

- Alternative approach implemented in Lab 2:
  - A **hardware timer** configured in CTC mode with interval *p*, triggers ISR every *p* ticks:

```
cli();                      /* Temporarily disable interrupts */
TCCR1A = 0;                 /* No output compare */
TCCR1B = (1 << WGM12);      /* CTC mode: reset timer when TCNT1 == OCR1A */
OCR1A = 0;                  /* Set to zero initially, over-wtite in ISR */
TCCR1B |= (1 << CS12);      /* Prescaler 256 (illustrative only) */
TIMSK1 |= (1 << OCIE1A);    /* Interrupt to call ISR when TCNT1 == OCR1A */
sei();                      /* Re-enable interrupts */
```

(**don’t learn details** but understand that timer triggers interrupt calling the ISR every period p)

Time per step: Implementation 2

- This **hardware timer** triggers an interrupt which is serviced by an ISR, which makes step & recalculates time *p* per step

```
ISR(TIMER1_COMPA_vect)
/* Interrupt service routine which calls moveOneStep and computeNewSpeed. */
{
  if (p == 0)
    TIMSK1 &= !(1 << OCIE1A); /* Disable interrupt if not stepping */
  else
    moveOneStep();

  OCR1A = (long)p - 1; /* Set timer (CTC) interval which is p ticks */
  computeNewSpeed(); /* Calculates timer interval p set in next ISR call */
}
```

Actually, make a step pulse

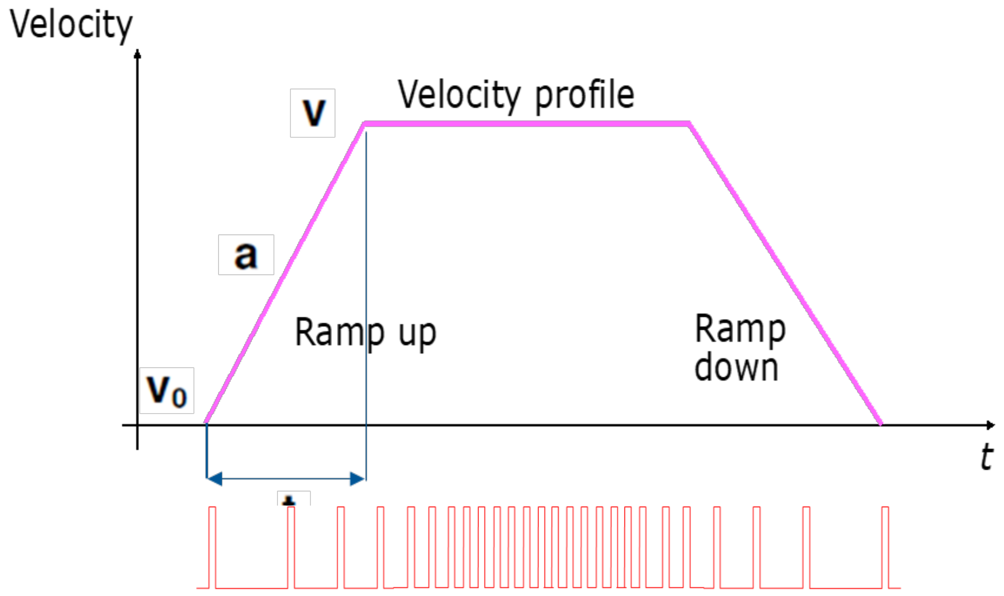
New interval *p* written to timer

p is re-calculated here for next step

Stepper Motors Control

Time per step (p) calculation

Time per step: possible solutions



The linear acceleration (ramping) formulas are:

$$S = v_0 \cdot t + a \cdot t^2 / 2 \qquad [1],$$

$$v = v_0 + a \cdot t \qquad [2]$$

where

- **S** - acceleration distance, in stepper motor case - **number of steps**,
- v_0 - initial velocity, **base speed** (steps per second),
- **v** - target velocity, **slew speed** (steps per second),
- **a** - **acceleration** (steps per second per second),
- **t** - acceleration time, **ramping period** (seconds).

By rearranging [2]

$$t = (v - v_0) / a \qquad [3]$$

and putting it in [1] we have

$$S = (v^2 - v_0^2) / (2 \cdot a) \qquad [4]$$

and

$$v = (v_0^2 + 2 \cdot a \cdot S)^{1/2} \qquad [5]$$

that can be represented as a **recursive form** of speed calculation for **one step**:

$$v_i = (v_{i-1}^2 + 2 \cdot a)^{1/2} \qquad [6]$$

where

i - step number $(1 \leq i \leq S)$.

2. Control basics

To produce the speed profile for stepper motor we need to provide the real time delays between step pulses:

$$p_i = F / v_i \qquad [7]$$

where

- p_i - delay period for the **i**-th step (timer ticks),
- **F** - **timer frequency** (count of timer ticks per second),

so according to [6] the exact delay value will be:

$$p_i = F / ((F / p_{i-1})^2 + 2 \cdot a)^{1/2} \qquad [8]$$

or

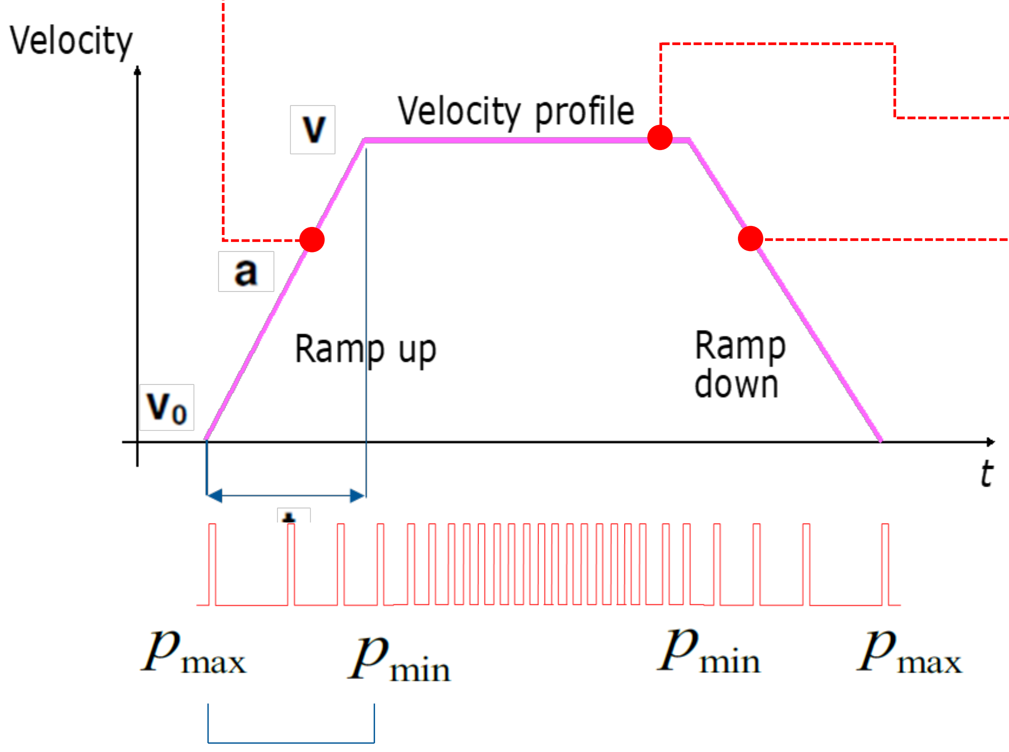
$$p_i = p_{i-1} / (1 + p_{i-1}^2 \cdot 2 \cdot a / F^2)^{1/2} \qquad [9].$$

?!

Time per step: a possible problem

- Calculating the next time interval *p* involves non-trivial calculations including division
- May be insufficient time available to calculate the time for the next step!
- So some ingenious methods have been derived which typically involve either:
  - Pre-calculation of an array of step times which can be executed at leisure, or:
  - Use approximate formulae
    1) Simple approximation
    2) Approximation based on Taylor series (e.g. Leib Ramp, Austin)

Approximate formulae: 1) Simple approximation



S: Number of steps

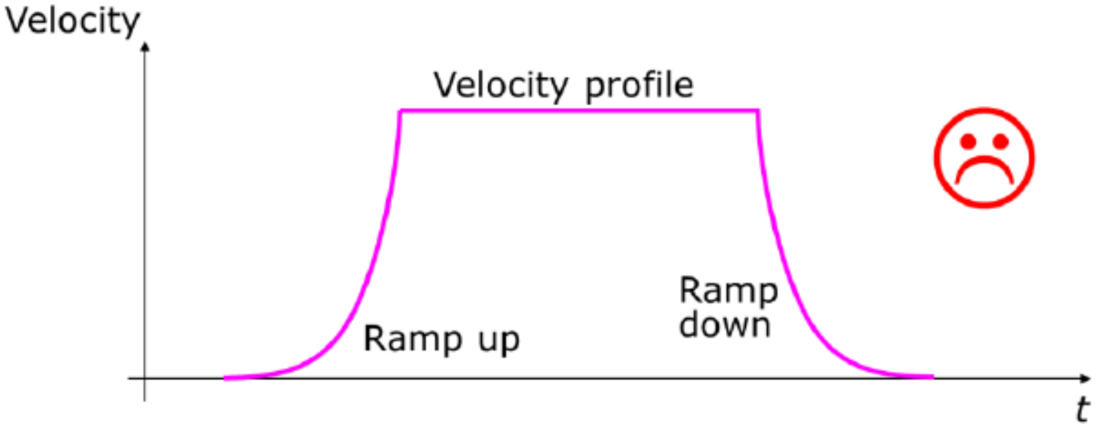
$$\Delta p = \frac{p_{\max} - p_{\min}}{S}$$

$$p_{new} = p_{old} - \Delta p \text{ (during acceleration)}$$

$$p_{new} = p_{old} \text{ (during constant speed phase)}$$

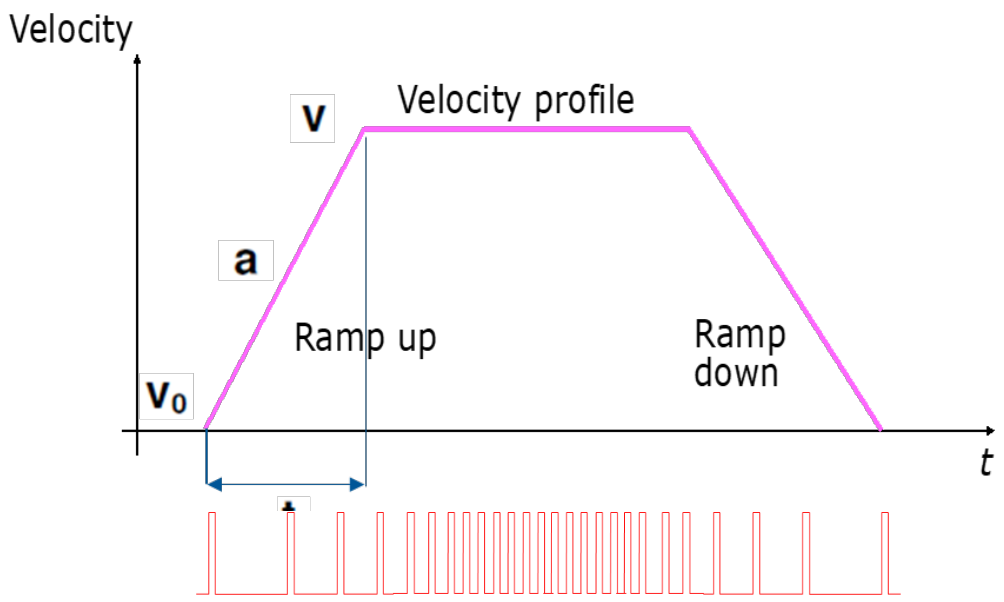
$$p_{new} = p_{old} + \Delta p \text{ (during deceleration)}$$

This will work but it is not that great profile ☹!



Approximate formulae: 2) Approximation based on Taylor series

Eiderman's Leib Ramp algorithm



$$p_i = p_{i-1} / (1 + p_{i-1}^2 \cdot 2 \cdot a / F^2)^{1/2} \quad [9].$$

?!

Using Taylor series

$$1 / (1 + n)^{1/2} \simeq 1 - n / 2 \quad [10]$$

when $-1 < n \leq 1$ we can approximate [9] to

$$p_i = p_{i-1} \cdot (1 - p_{i-1}^2 \cdot a / F^2) \quad [11].$$

Let's check the $-1 < n \leq 1$ condition. Our n was

$$n = p_{i-1}^2 \cdot 2 \cdot a / F^2 \quad [12]$$

or, by velocity,

$$n = 2 \cdot a / v_{i-1}^2 \quad [13].$$

from $p_i = F / v_i$ [7]

The maximum n value will be at minimum speed, on the first calculated step, where $i = 2$

$$n_{max} = 2 \cdot a / v_1^2 \quad [14].$$

Because the minimal v_0 is 0, from [6] we have

$$v_{1min} = (2 \cdot a)^{1/2} \quad [15].$$

So n will be **always** less than or equal to 1. Because our calculations are forward-only we have no limitation in case of deceleration (negative acceleration) too.

$$R = a / F^2 \quad [19].$$

$m = -R$ during acceleration phase,
$m = 0$ between acceleration and deceleration phases,
$m = R$ during deceleration phase.

The variable delay period p (initially $p = p_1$) that will be recalculated for each next step is:

$$p = p \cdot (1 + m \cdot p \cdot p) \quad [20].$$

Using the higher order approximation of Taylor series

$$1 / (1 + n)^{1/2} \simeq 1 - n / 2 + 3 \cdot n^2 / 8 \quad [21]$$

we can get more accurate results replacing [20] with

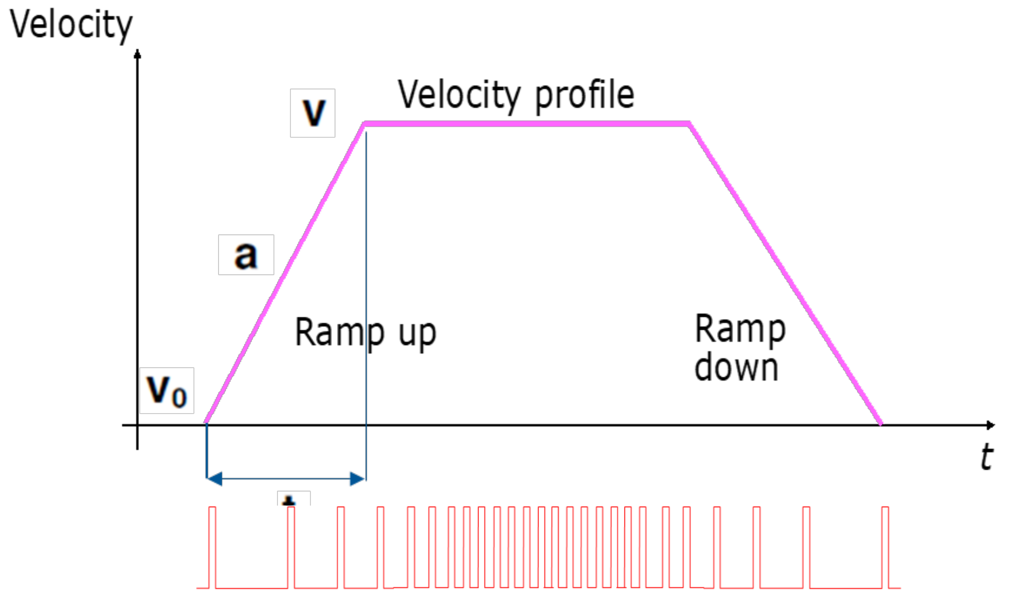
$$p = p \cdot (1 + q + 1.5 \cdot q \cdot q) \quad [22]$$

where

$$q = m \cdot p \cdot p .$$

Optional enhancement

Approximate formulae: 2) Approximation based on Taylor series



The motion profile inputs

v_0 - initial velocity, **base speed** (steps per second),
v - target velocity, **slew speed** (steps per second),
a - **acceleration** (steps per second per second),
t - acceleration time, **ramping period** (seconds).

Up-front calculation – Time-intensive

S - acceleration/deceleration distance

$$S = (v^2 - v_0^2) / (2 \cdot a) \qquad [4, 16],$$

p_1 - delay period for the **initial** step

$$p_1 = F / (v_0^2 + 2 \cdot a)^{1/2} \qquad [17],$$

p_S - delay period for the **slew speed** steps

$$p_S = F / v \qquad [18],$$

R - constant multiplier

$$R = a / F^2 \qquad [19].$$

p can be calculated on-the-fly

$$p_{\text{new}} = p_{\text{old}}\left(1 + q + 1.5 \times q \times q\right)$$

$$q = m \times p_{\text{old}} \times p_{\text{old}}$$

$$m = -R \quad \text{or} \quad m = 0 \quad \text{or} \quad m = -R$$

The calculations are simple
(addition and multiplication) ☺!

Stepper motor control on Arduino

- Not usually appropriate to write your own…
- Various libraries available of varying usefulness:
  - **Stepper** library comes as standard: only allows constant-speed movement, no ramping.
  - Drives H bridge (e.g L298) directly, not compatible with "step and direction" drivers used industrially and in labs
  - "Blocking" i.e. can't run steps in background and get on with other tasks

Stepper motor control on Arduino

- Various libraries available of varying usefulness:
  - **AccelStepper**: broadly similar to approach used in Lab 2
  - Uses timed loop based on **micros()**
  - Approximate calcs based on Taylor series
  - Based on theory by Austin (rather than Eiderman's Leib Ramp
  - Timed loop called repeatedly from **loop()**
  - Non-blocking: can get on with other tasks
  - Forms basis of various other libraries

University of Nottingham
UK | CHINA | MALAYSIA

Stepper Motor Characteristics

Introduction

Stepper motor characteristics

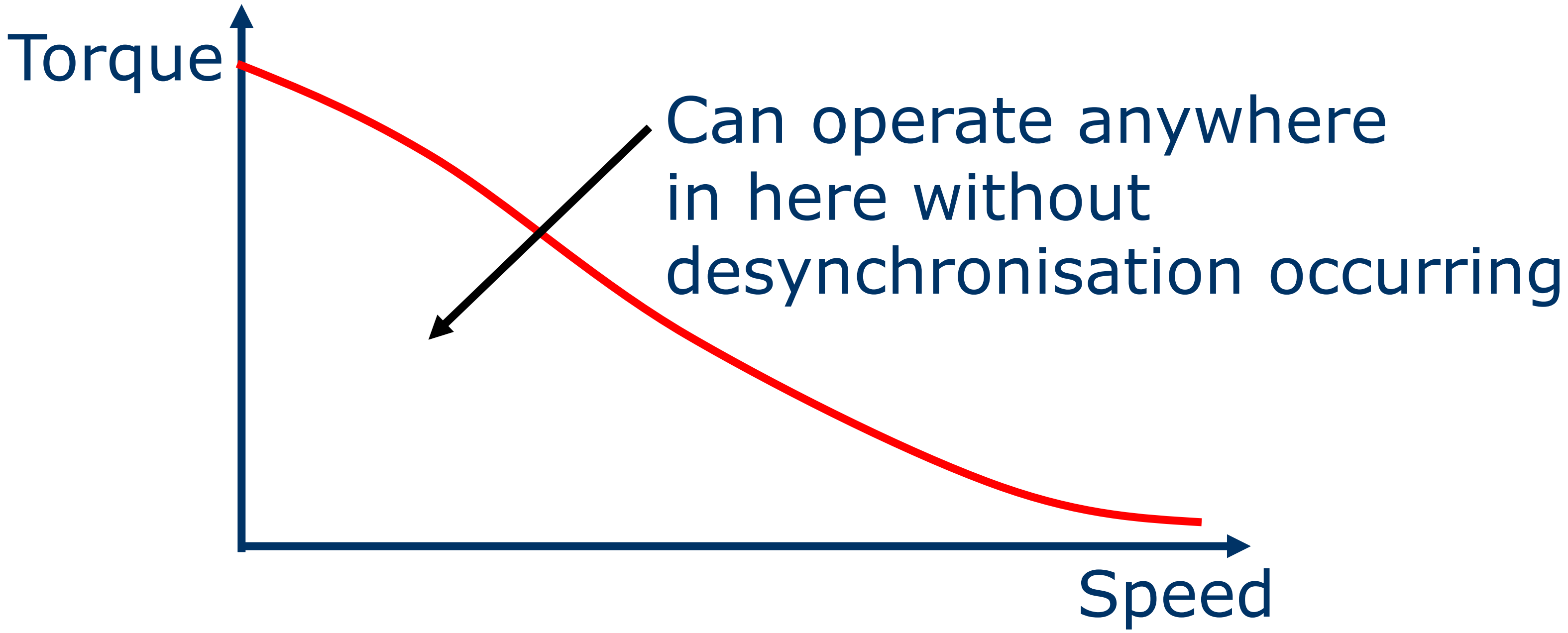
- Stepper motors run at a speed directly proportional to the step rate
- Stepper motors don’t work like that!
- As the load varies:
  - Either the motor runs as planned at the correct speed for that pulse rate
  - Or the motor stops running as planned and becomes desynchronised

Stepper motor characteristics

- Generally speaking, if motor becomes desynchronised with the magnetic field driving it around:
  - Position and accuracy are lost for remainder of the time machine is in use before being reset
  - Whole purpose of stepper motor is negated

Stepper motor characteristics

- Typical characteristics are quoted for a given motor supply voltage or current



University of Nottingham
UK | CHINA | MALAYSIA

Stepper Motor Characteristics

Dynamics

Stepper Motor Dynamics

- To select a stepper motor need to know:
  - what is the inertia of the driven system, referred to the stepper motor axis?
  - NB: driven inertia should be similar to that of motor itself (not hugely larger) or dynamics will be poor
  - what is the speed profile required?
  - what is the acceleration/deceleration required?
  - hence what torque is required from the motor?

Stepper Motor Characteristics

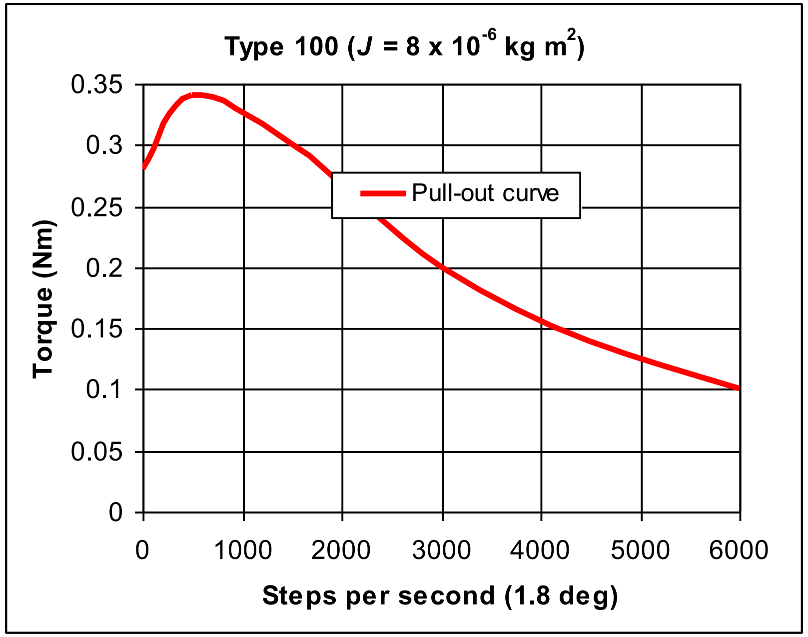
Dynamics, Example

Stepper motor dynamics: Example

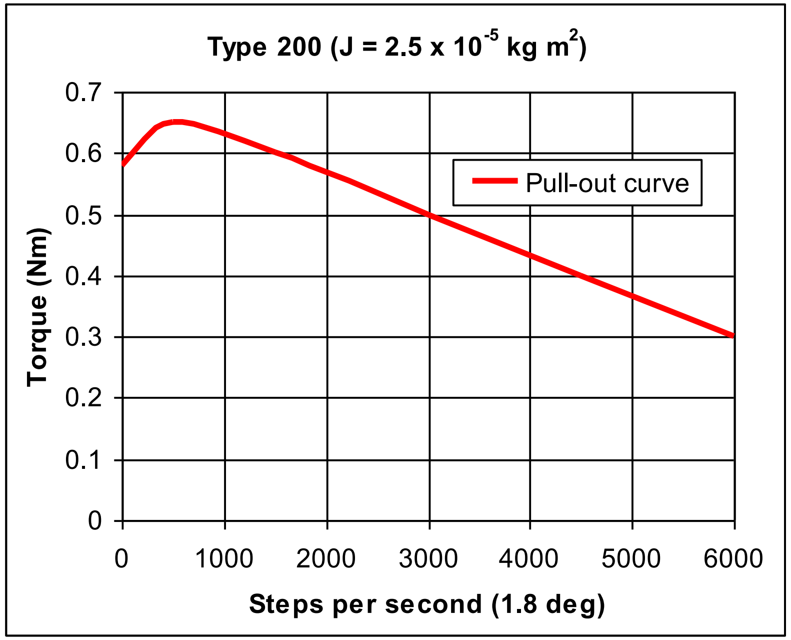
- A system has a moment of inertia (referred to the stepper motor shaft) of 6×10^{-5} kg m^2. Its torque-speed characteristic may be modelled as a constant frictional torque of 0.3 Nm.
- It needs to be accelerated from rest to its maximum speed of 20 rev/s in 0.2 second. Select a stepper motor from the range (call them types 100, 200, 300) whose characteristics are given in the slides.

Stepper motor dynamics: Example

Type 100



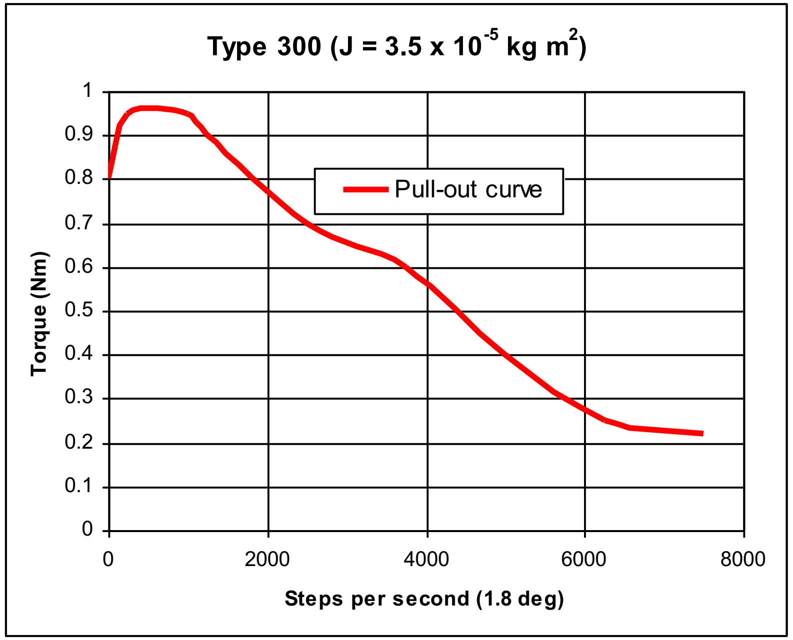
Type 200



Very loosely based on the McLennan stepper motor characteristics

University of Nottingham
UK | CHINA | MALAYSIA

Type 300



Very loosely based on the McLennan stepper motor characteristics

Stepper motor dynamics: Example

In practice you would have to calculate the frictional torque, maximum speed, acceleration and referred inertia yourself from the characteristics and desired behaviour of the system. For simplicity, we've done it for you

Stepper motor dynamics: Example

- Maximum speed is 20 rev/s. We need to convert this to steps/s
- Each step is 1.8 degrees
- There are 360 degrees in one rev.

- So 20 rev/s is $20 \times 360/1.8 = 4000$ steps per second

Stepper motor dynamics: Example

- Acceleration: for dynamics we need this in rad/s2
- Maximum speed is 20 rev/s which is:

$$20 \times 2\pi = 40\pi \text{ rad/s}$$

- Go from zero to 40π rad/s in 0.2s
- Acceleration is 40/0.2 rev/s² = 100 rev/s2

$$= 200\pi$$

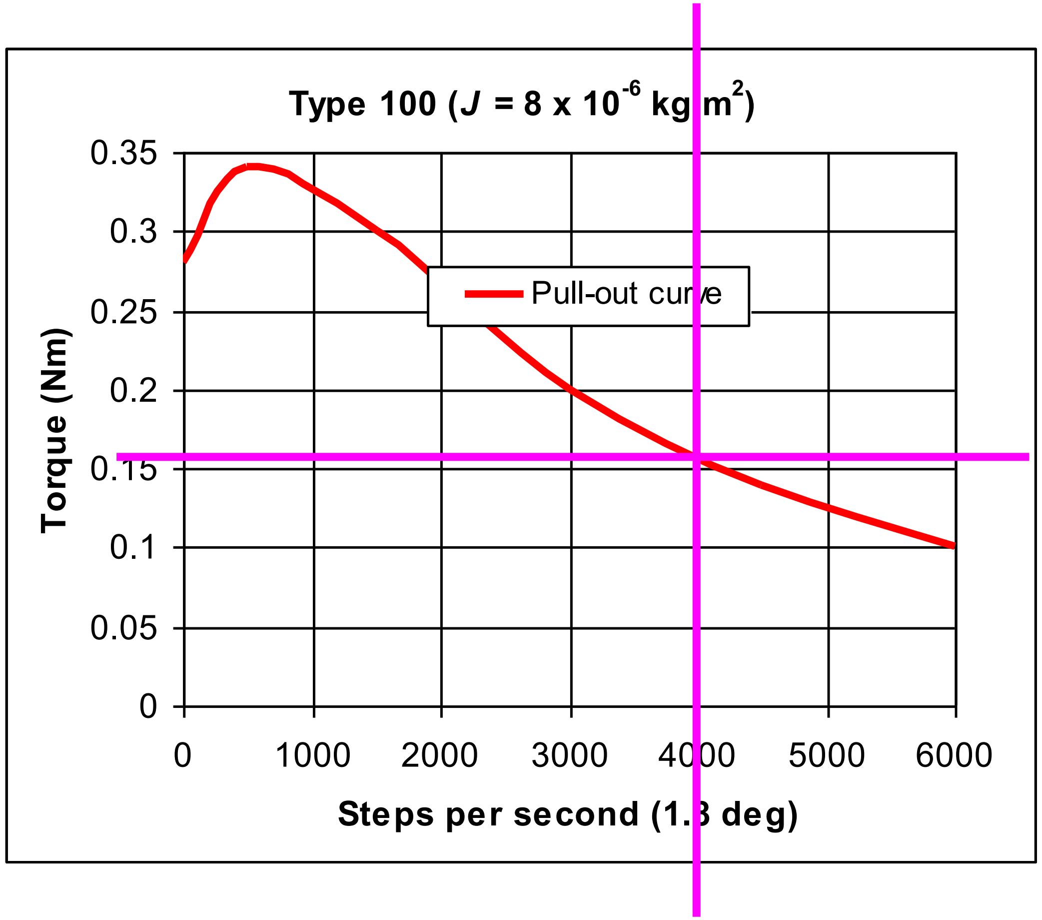
$$= 628.3 \text{ rad/s2}$$ (call this α)

Stepper motor dynamics: Example

- First check: can motor produce enough steady-state torque at required speed?
- Remember, we need 0.3 Nm at 4000 steps/s

- If we choose **type 100**, max torque at 4000 steps/s is about **0.15 Nm** so can see straight away that it cannot provide necessary torque even for constant speed running (0.3 Nm).

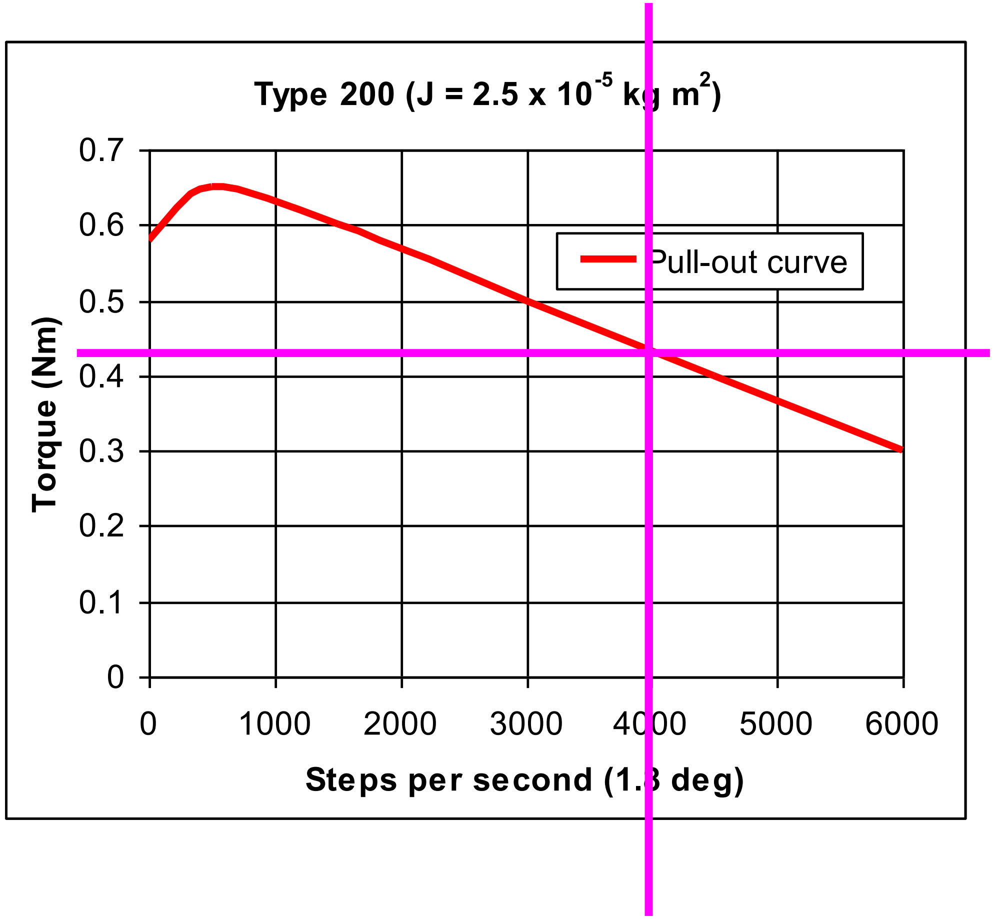
Stepper motor dynamics: Example

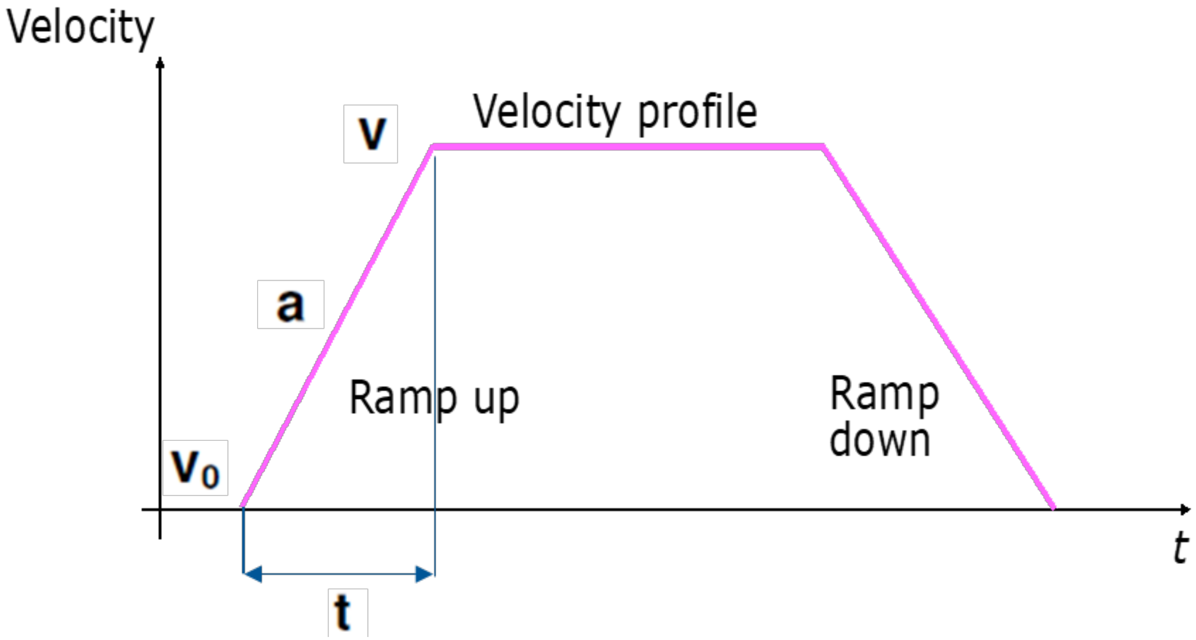


Stepper motor dynamics: Example

- If we choose type 200: can produce enough torque for steady-state running (can produce 0.43 Nm, need 0.3 Nm)
- But has it got enough torque to accelerate rapidly enough?
- Need enough torque to accelerate **whole system** at 628.3 rad/s^2 **AND** overcome friction

Stepper motor dynamics: Example





Stepper motor dynamics: Example

If we choose type 200:

has inertia 2.5×10^{-5} kg m^2,

total moment of inertia is $2.5\times10^{-5} + 6\times10^{-5}$

$= 8.5\times10^{-5}$ kg m^2 (call this J_{total})

Stepper motor dynamics: Example

Total torque required is:

torque to cause acce'n + steady state torque

$= J_{total} \times \alpha +$ steady-state torque

$= 8.5\times10^{-5}\times628.3 + 0.3 = 0.353$ Nm.

So, the available torque of 0.43 Nm is OK by a small factor (1.2) – not much margin for error

Stepper motor dynamics: Example

If we choose type 300:

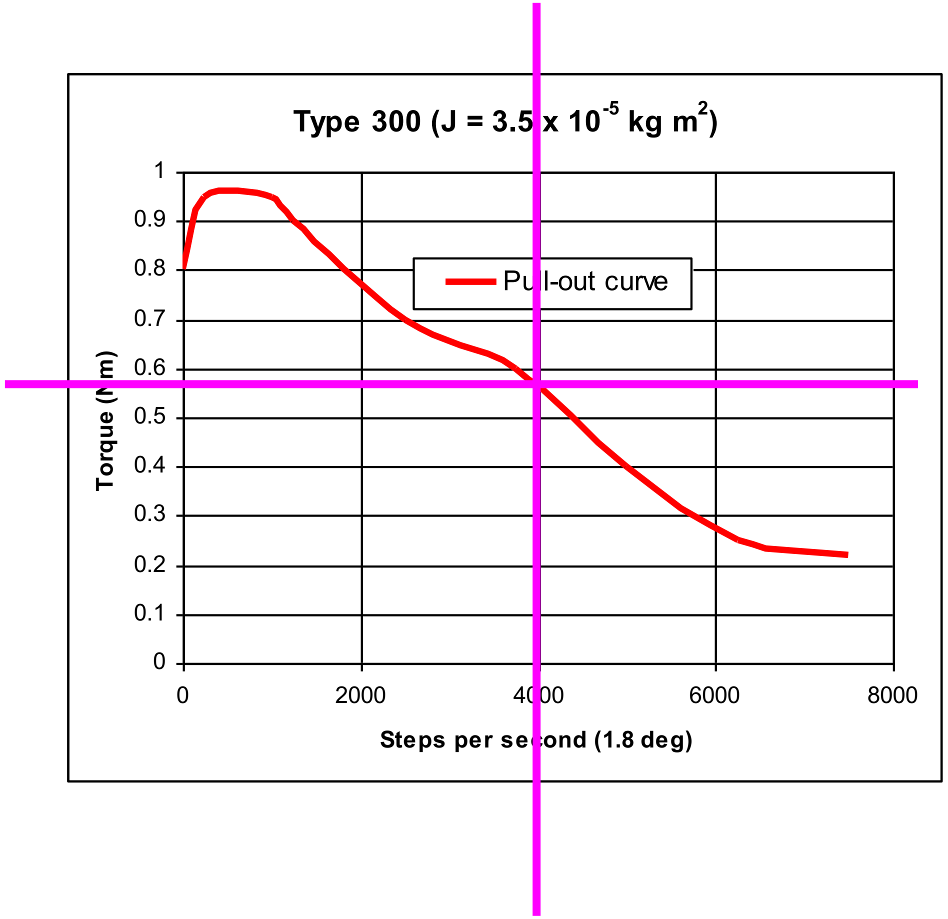
has inertia 3.5×10^{-5} kg m², total moment of inertia is 9.5×10^{-5} kg m².

Total torque required is:

$$9.5\times 10^{-5} \times 628.3 + 0.3 = 0.359 \text{ Nm}.$$

Torque available at 4000 steps/s is 0.57 Nm so probably a safer bet, factor of safety 1.61 (better, probably about the minimum).

Stepper motor dynamics: Example

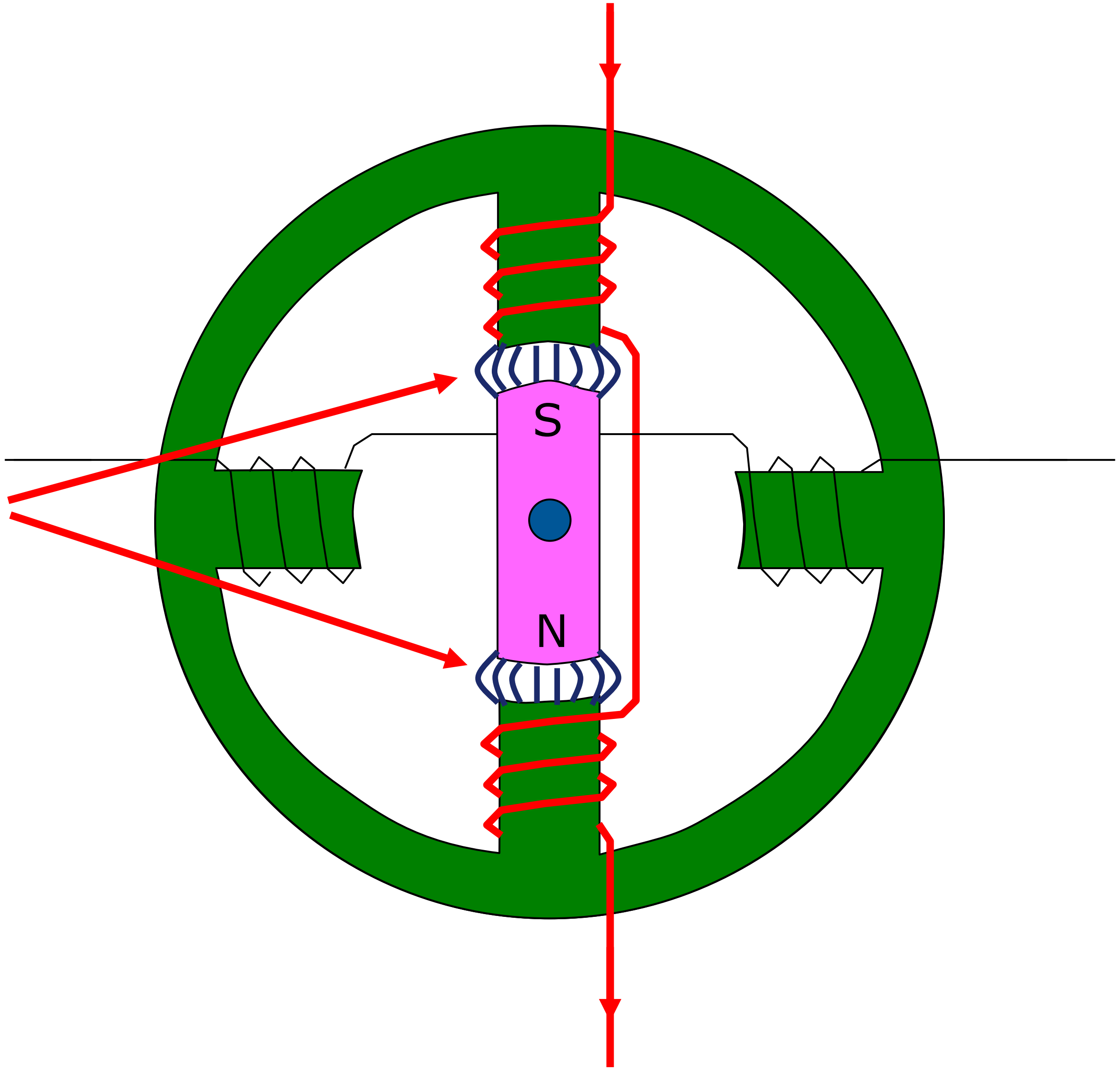


Stepper Motor Characteristics

Resonance

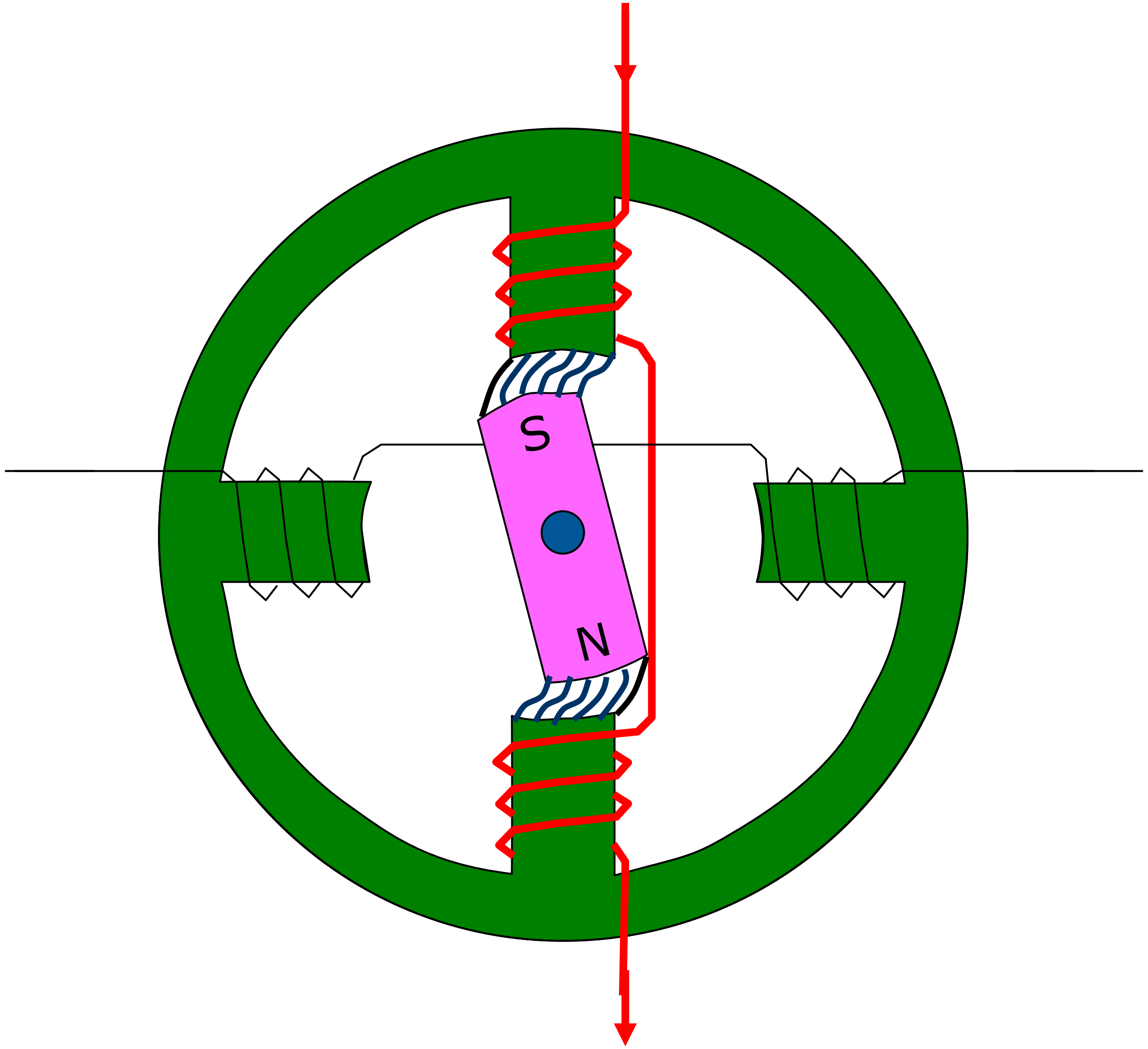
A major problem with stepper motors: resonance

- Rotor is held at a given position by magnetic flux
- Rotor is not held rigidly - has a finite stiffness



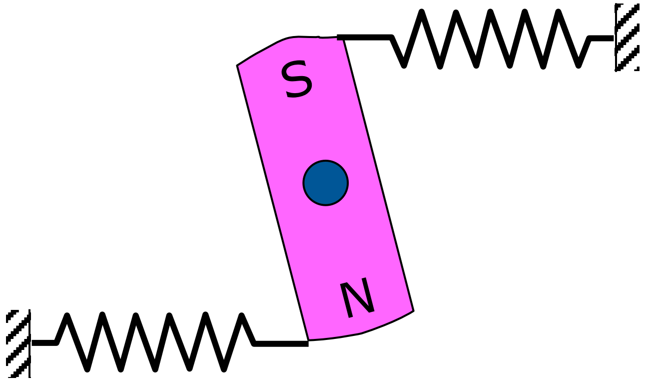
A major problem with stepper motors: resonance

- When torque is applied, rotor can be displaced from its neutral position
- But remember the rotor also has inertia



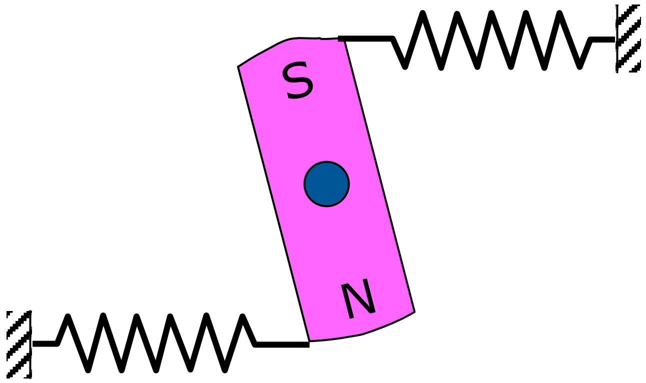
A major problem with stepper motors: resonance

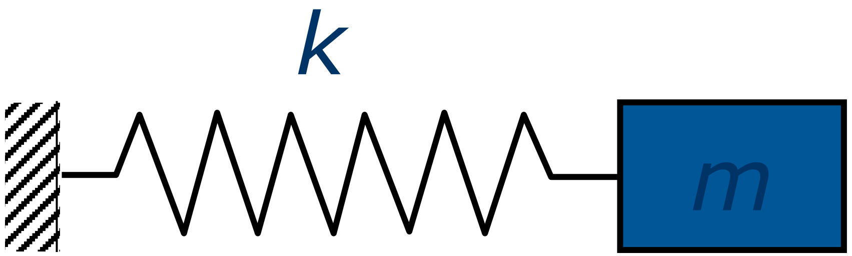
- When torque is applied, rotor can be displaced from its neutral position
- But remember the rotor also has inertia



A major problem with stepper motors: resonance

- So, we have a system with:
  - rotational (angular) stiffness
  - rotational inertia
- Rotational version of mass-spring system (no obvious damping!)



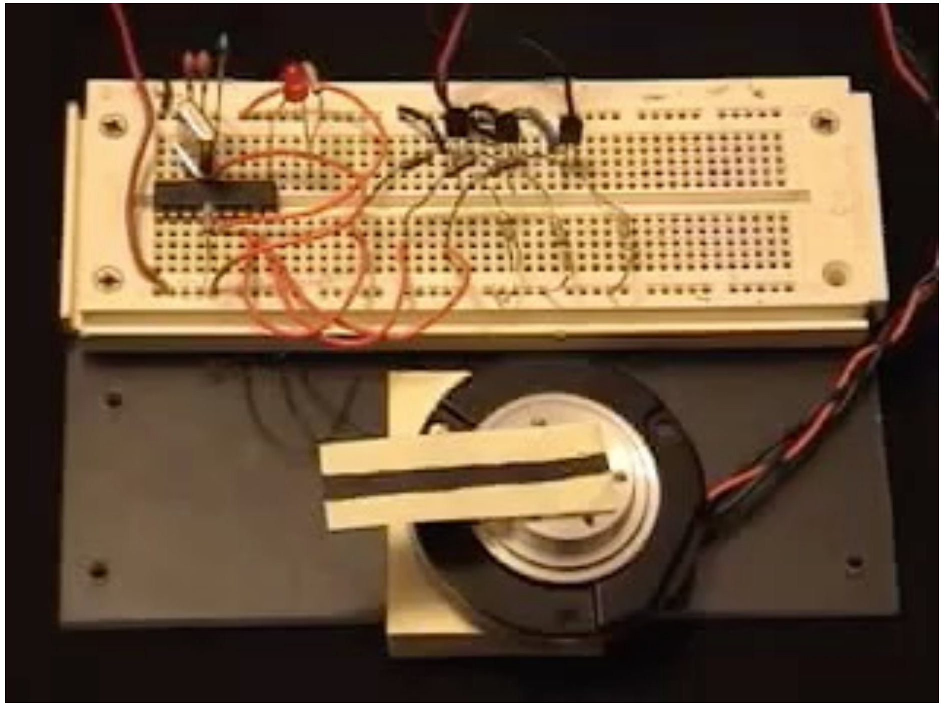


A major problem with stepper motors: resonance

- Inertia + stiffness + some oscillatory driving force (e.g. stepping) = **resonance**
- At resonance, rotor oscillates instead of stepping neatly from pole to pole
- Loss of synchronisation hence loss of usefulness of system

A major problem with stepper motors: resonance

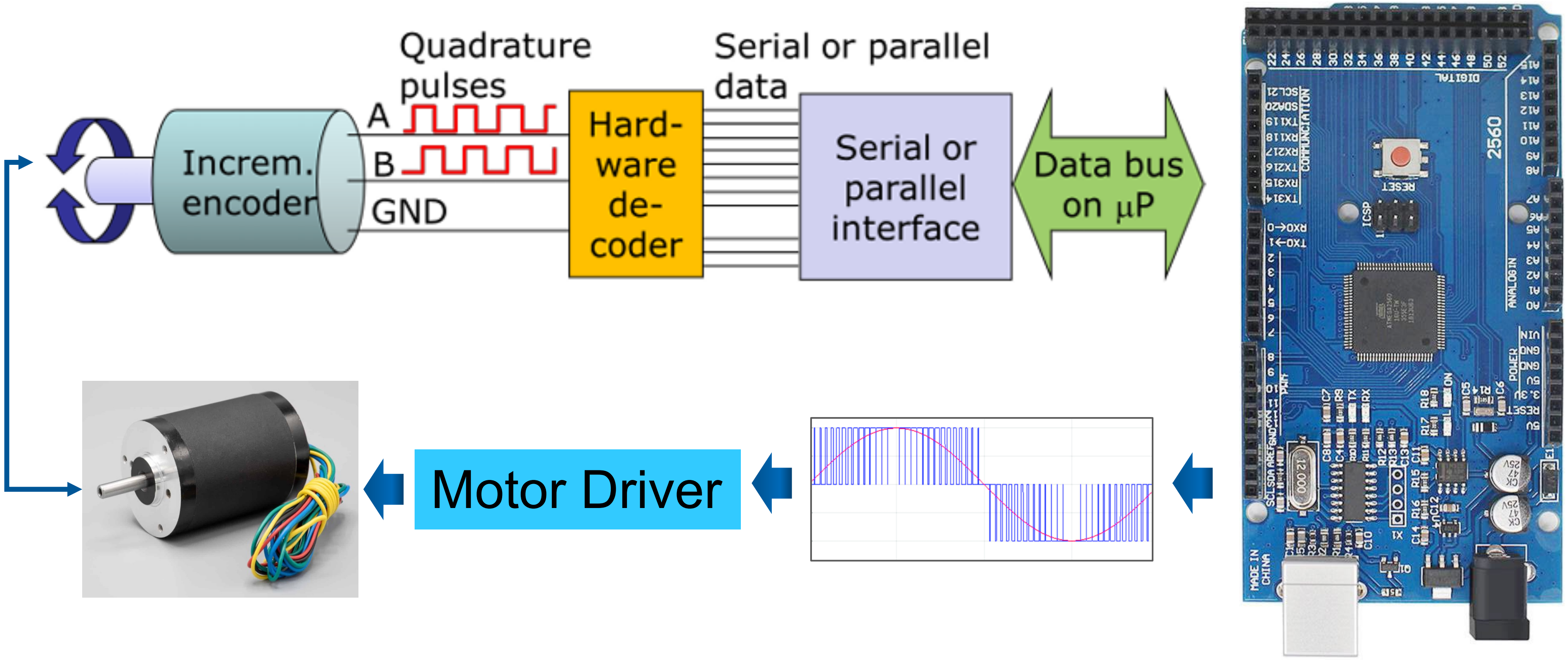
- Demonstration of resonance



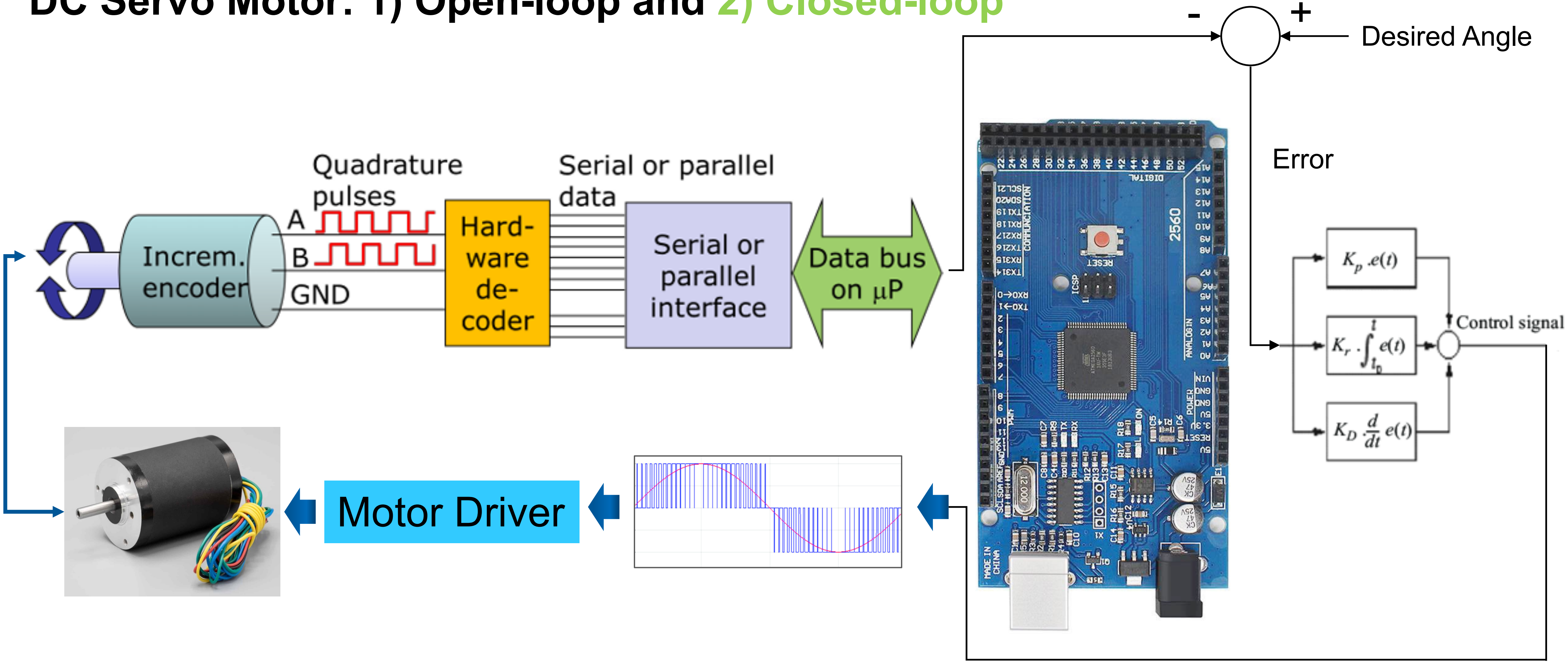
- Stepper motor behaving badly!

Link Lab 2 to Lectures

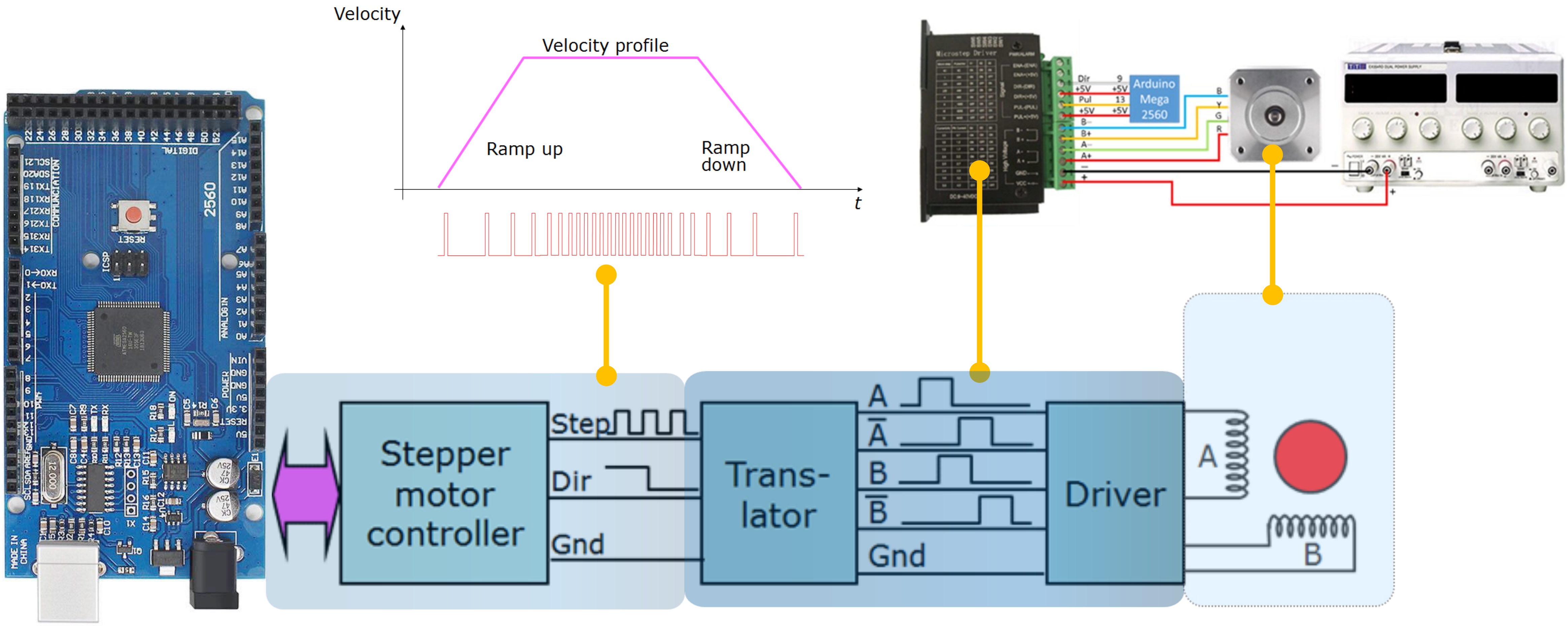
DC Servo Motor: 1) Open-loop and 2) Closed-loop



DC Servo Motor: 1) Open-loop and 2) Closed-loop

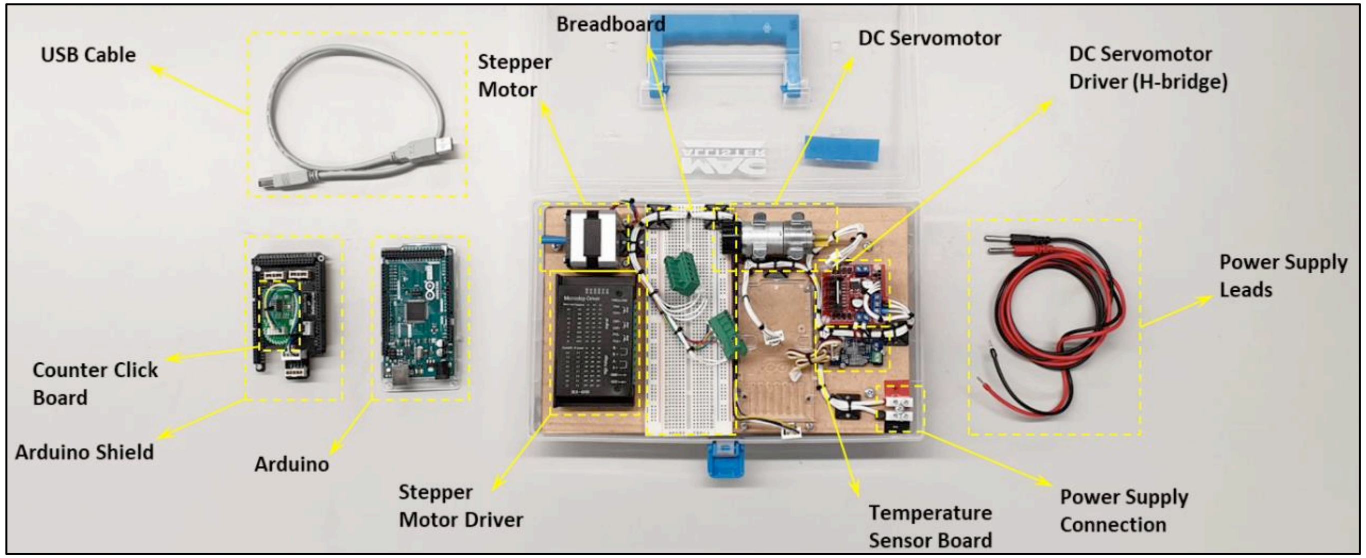


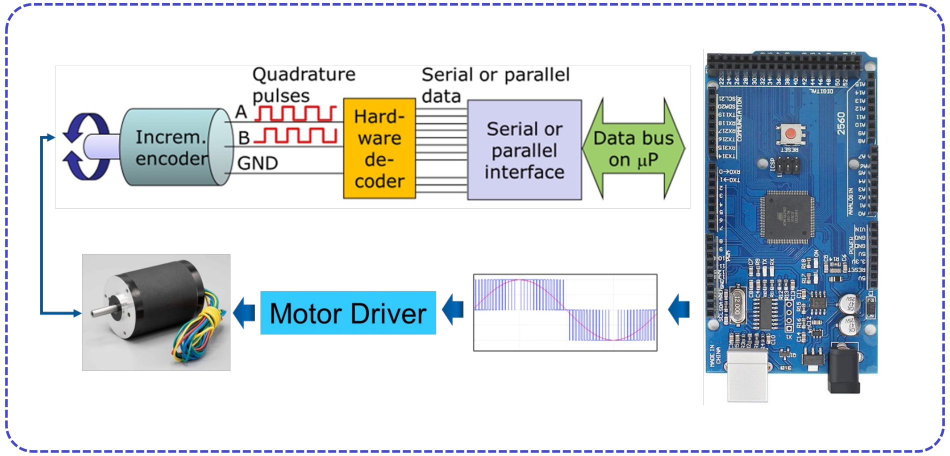
Stepper Motor: Open-loop



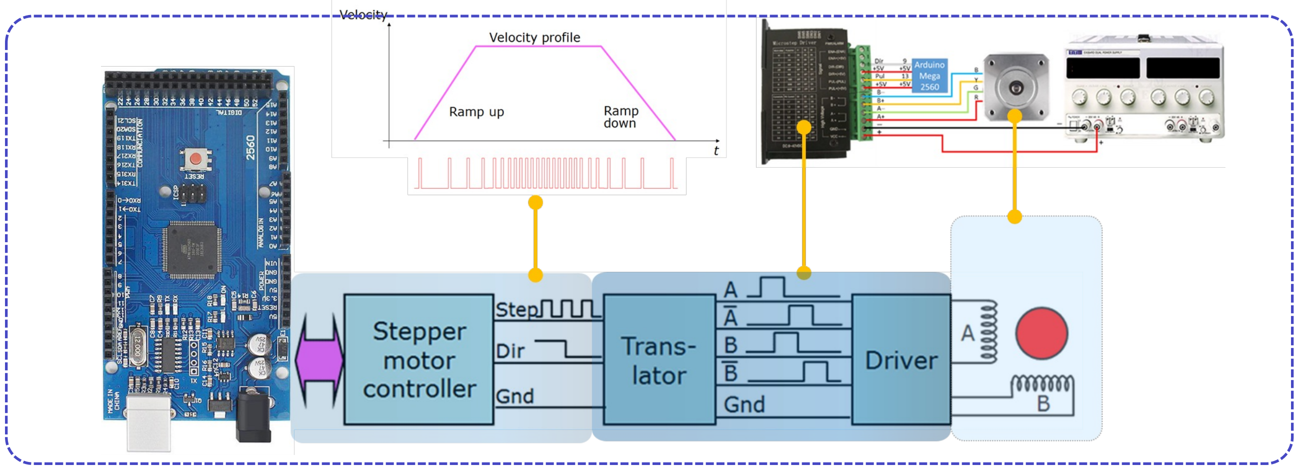
Laboratory 2: Motion Control

- This is the lab kit which you will use in Lab 2 (same as lab 1)

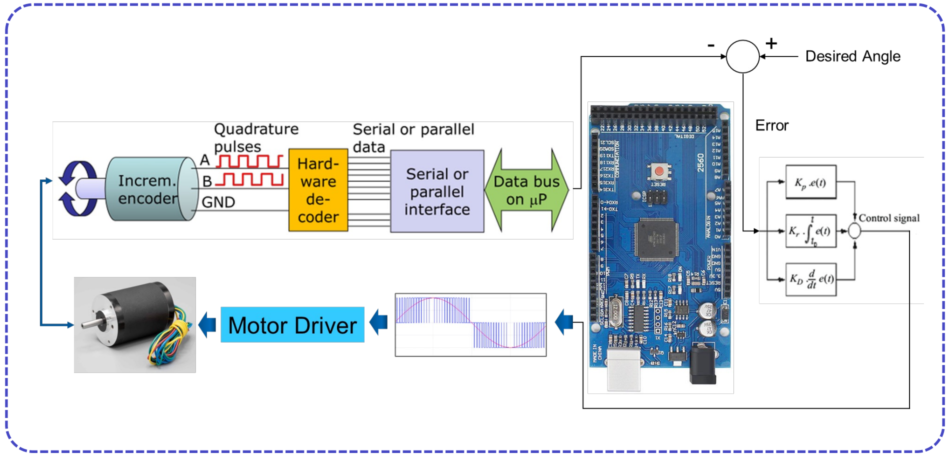




Experiment 1: DC Servo Motor: Open-loop



Experiment 3: Stepper Motor (Open-loop)

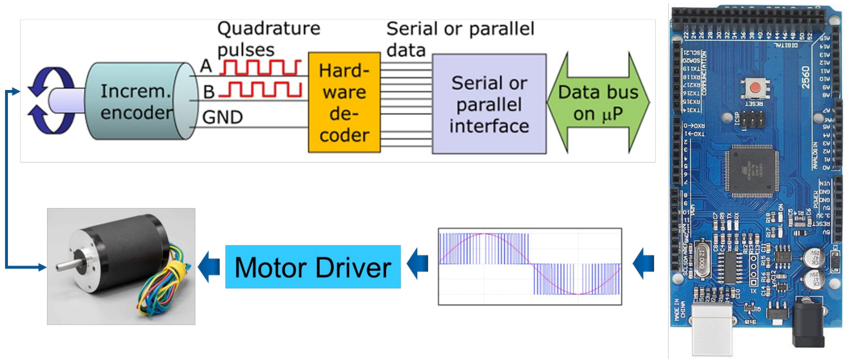


Experiment 2: DC Servo Motor: Closed-loop

University of Nottingham
UK | CHINA | MALAYSIA

Have a Look Into the Lab Code!

Experiment 1: DC Motor (Open-loop control)



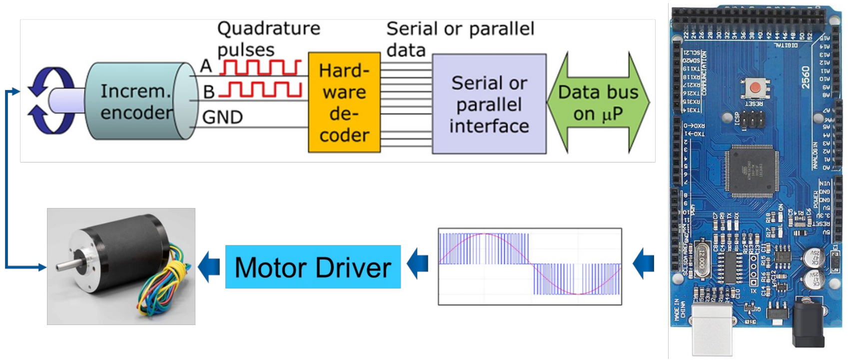

```
/* Pins used for L298 DC Motor driver */
#define enA 13       /* PWM output, also visible as LED */
#define in1 8        /* H bridge selection input 1 */
#define in2 9        /* H bridge selection input 2 */
#define minPercent -100.0
#define maxPercent 100.0
```

```
void setup()
{
  Serial.begin(9600);
  Serial.println("Enter PWM duty cycle as a percentage (positive for forward, negative for reverse");

  /* Set up and initialise pin used for selecting LS7366R counter: hi=inactive */
  pinMode(chipSelectPin, OUTPUT);
  digitalWrite(chipSelectPin, HIGH);

  SetUpLS7366RCounter();

  delay(100);

  /* Configure control pins for L298 H bridge */
  pinMode(enA, OUTPUT);
  pinMode(in1, OUTPUT);
  pinMode(in2, OUTPUT);

  /* Set initial rotation direction */
  digitalWrite(in1, LOW);
  digitalWrite(in2, HIGH);

}
```

```
void driveMotorPercent(double percentSpeed)
/* Output PWM and H bridge signals based on positive or negative duty cycle % */
{
    percentSpeed = constrain(percentSpeed, -100, 100);   // Value must be in range -100 to +100
    int regVal = map(percentSpeed, -100, 100, -255, 255);   // Scale value to range -255 to +255
    analogWrite(enA, (int)abs(regVal));      // Write value to speed control pin
    digitalWrite(in1, regVal>0);             // Set the value of direction control pins to true or false
    digitalWrite(in2, !(regVal>0));          // depending on whether speed is positive or negative
}
```

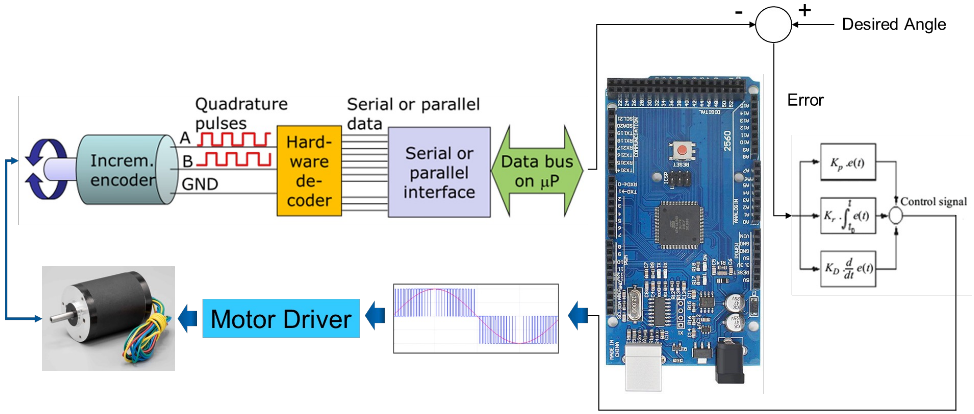
```
void printLoop()
/* Print count and control information */
{
  /* Sample counter chip and output position and requested speed */
  long encoderCountFromLS7366R = readEncoderCountFromLS7366R();

  Serial.print("Count from LS7366R = ");
  Serial.print(encoderCountFromLS7366R);
  Serial.print(" Percent speed = ");
  Serial.print(percentSpeed);
  Serial.print("\r\n");
}
```

Experiment 2: DC Motor (Closed-loop control)




```
void loop()
{
  unsigned long currentMillis = millis();

  // Print out value to serial monitor at interval specified by printInterval variable
  if (currentMillis - prevMillisPrint >= printInterval) {
    // save the last time you printed output
    prevMillisPrint = currentMillis;
    printLoop();
  }

  // Check if new data has been input via serial monitor
  recvWithEndMarker();
  if(convertNewNumber())  // Update value read from serial line
  {
    percentSpeed=dataNumber;
    driveMotorPercent(percentSpeed);  // Send new speed value to motor driver
  }
}
```

Receive *speed* from the user

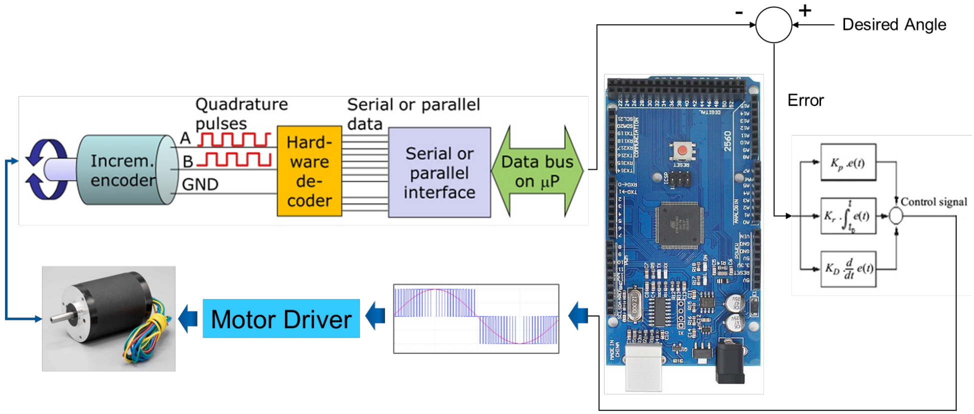



```
void loop()
{
  unsigned long currentMillis = millis();

  // Call control loop at frequency controInterval
  if (currentMillis - prevMillisControl >= controlInterval)
  {
    // Save the current time for comparison the next time the loop is called
    prevMillisControl = currentMillis;
    controlLoop();
  }

  // Call print loop at frequency of printInterval
  if (currentMillis - prevMillisPrint >= printInterval)
  {
    // Save the current time for comparison the next time the loop is called
    prevMillisPrint = currentMillis;
    printLoop();
  }

  recvWithEndMarker();  // Update value read from serial line
  // If a valid number has been read this is set to the current required position
  if(convertNewNumber())
  {
    positionSetPoint = dataNumber;
  }
}
```

Receive *position* from the user

Experiment 2: DC Motor (Closed-loop control)

Proportional Controller (P)



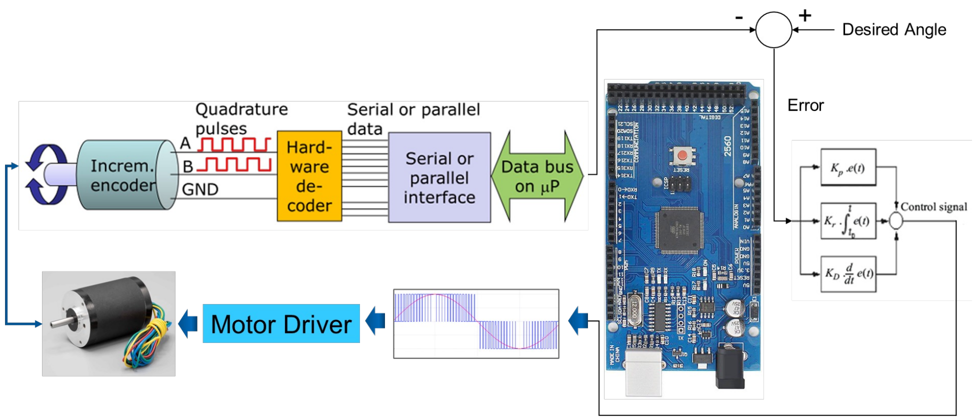

```
void controlLoop()
{
    double error;
    // Get the current position from the encoder
    encoderPosnMeasured = readEncoderCountFromLS7366R();
    // Calculate the difference in position from the required position
    error = positionSetPoint - (double)encoderPosnMeasured;
    // Multiply by the gain
    percentDutyCycle = error * Kp;
    driveMotorPercent(percentDutyCycle);
}
```

```
void loop()
{
  unsigned long currentMillis = millis();

  // Call control loop at frequency controInterval
  if (currentMillis - prevMillisControl >= controlInterval)
  {
    // Save the current time for comparison the next time the loop is called
    prevMillisControl = currentMillis;
    controlLoop();
  }

  // Call print loop at frequency of printInterval
  if (currentMillis - prevMillisPrint >= printInterval)
  {
    // Save the current time for comparison the next time the loop is called
    prevMillisPrint = currentMillis;
    printLoop();
  }

  recvWithEndMarker();  // Update value read from serial line
  // If a valid number has been read this is set to the current required position
  if(convertNewNumber())
  {
    positionSetPoint = dataNumber;
  }
}
```

Receive *position* from the user

Experiment 2: DC Motor (Closed-loop control)

Proportional Integral and Derivative Controller (PID)




```
void controlLoop()
{
    double error;
    // Get the current position from the encoder
    encoderPosnMeasured = readEncoderCountFromLS7366R();
    // Calculate the difference in position from the required position
    error = positionSetPoint - (double)encoderPosnMeasured;
    // Multiply by the gain
    percentDutyCycle = error * Kp;
    driveMotorPercent(percentDutyCycle);
}
```

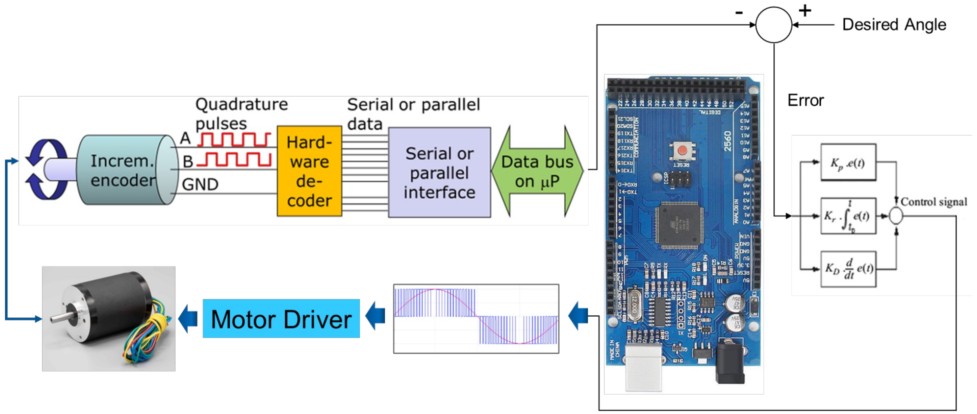
```
void controlLoop()
{
  // Get the current position from the encoder
 encoderPosnMeasured=readEncoderCountFromLS7366R();  // Get current motor position
 myPID.Compute();          // Use the PID library to compute new value for motor input
 driveMotorPercent(percentSpeed);    // Send value to motor
}
```

```
void loop()
{
  unsigned long currentMillis = millis();
  // Call control loop at frequency controInterval
  if (currentMillis - prevMillisControl >= controlInterval)
  {
    // Save the current time for comparison the next time the loop is called
    prevMillisControl = currentMillis;
    controlLoop();
  }
  // Call print loop at frequency of printInterval
  if (currentMillis - prevMillisPrint >= printInterval)
  {
    // Save the current time for comparison the next time the loop is called
    prevMillisPrint = currentMillis;
    printLoop();
  }
  recvWithEndMarker();  // Update value read from serial line
  // If a valid number has been read this is set to the current required position
  if(convertNewNumber())
  {
    positionSetPoint = dataNumber;
  }
}
```

Receive *position* from the user

Experiment 2: DC Motor (Closed-loop control)

Proportional Integral and Derivative Controller (PID)




```
void controlLoop()
{
  // Get the current position from the encoder
  encoderPosnMeasured=readEncoderCountFromLS7366R();  // Get current motor position
  myPID.Compute();           // Use the PID library to compute new value for motor input
  driveMotorPercent(percentSpeed);     // Send value to motor
}
```

```
void loop()
{
  unsigned long currentMillis = millis();
  // Call control loop at frequency controInterval
  if (currentMillis - prevMillisControl >= controlInterval)
  {
    // Save the current time for comparison the next time the loop is called
    prevMillisControl = currentMillis;
    controlLoop();
  }
  // Call print loop at frequency of printInterval
  if (currentMillis - prevMillisPrint >= printInterval)
  {
    // Save the current time for comparison the next time the loop is called
    prevMillisPrint = currentMillis;
    printLoop();
  }
  recvWithEndMarker();  // Update value read from serial line
  // If a valid number has been read this is set to the current required position
  if(convertNewNumber())
  {
    positionSetPoint = dataNumber;
  }
}
```

Receive *position* from the user

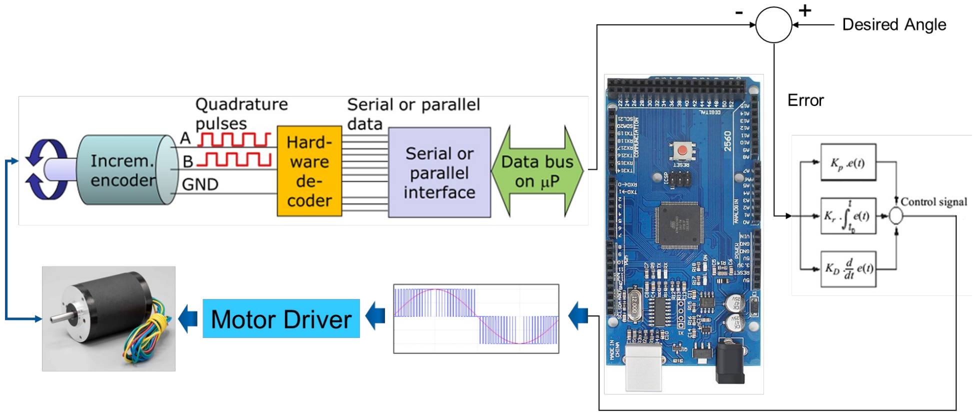
Experiment 2: DC Motor (Closed-loop control)

Proportional Integral and Derivative Controller (PID)

```
#include <PID_v1.h>
```

```
/* PID */
double Kp = 0.05;
double Ki = 0.0;
double Kd = 0.0;

PID myPID(&encoderPosnMeasured, &percentSpeed, &positionSetPoint, Kp, Ki, Kd, DIRECT);
```

```
void controlLoop()
{
  // Get the current position from the encoder
  encoderPosnMeasured=readEncoderCountFromLS7366R();  // Get current motor position
  myPID.Compute();          // Use the PID library to compute new value for motor input
  driveMotorPercent(percentSpeed);     // Send value to motor
}
```

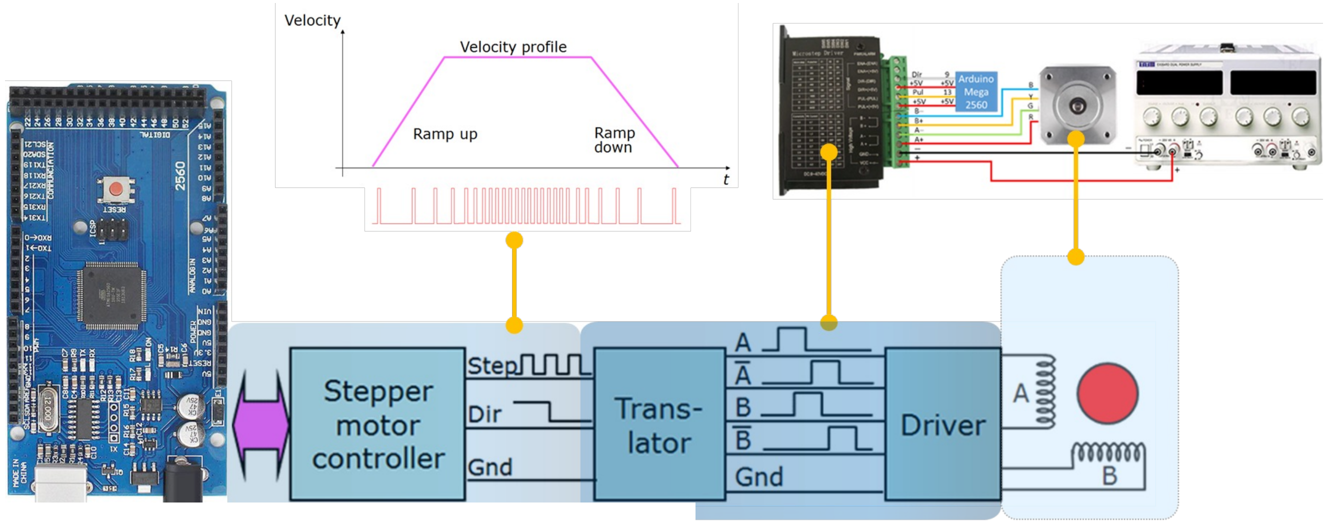



```
void loop()
{
  unsigned long currentMillis = millis();

  // Call control loop at frequency controInterval
  if (currentMillis - prevMillisControl >= controlInterval)
  {
    // Save the current time for comparison the next time the loop is called
    prevMillisControl = currentMillis;
    controlLoop();
  }

  // Call print loop at frequency of printInterval
  if (currentMillis - prevMillisPrint >= printInterval)
  {
    // Save the current time for comparison the next time the loop is called
    prevMillisPrint = currentMillis;
    printLoop();
  }

  recvWithEndMarker();  // Update value read from serial line
  // If a valid number has been read this is set to the current required position
  if(convertNewNumber())
  {
    positionSetPoint = dataNumber;
  }
}
```

Receive *position* from the user

Experiment 3: Stepper Motor (Open-loop control)

1) Simple approximation
2) Approximation based on Taylor series (e.g. Leib Ramp, Austin)



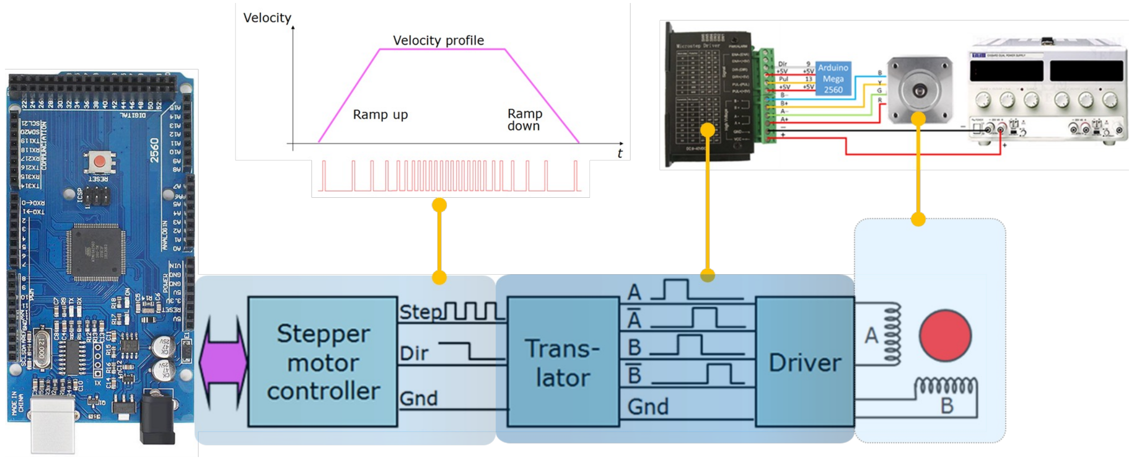

```
void setup()
{
    long stepsToGo = 0;
    currentPosition = 0;
    goToPosition(dataNumber);
    pinMode(stepPin, OUTPUT);
    pinMode(dirPin, OUTPUT);
    Serial.begin(9600);
    Serial.println("Enter target position in number of steps and hit return");

    prevStepTime = micros();
}
```

Experiment 3: Stepper Motor (Open-loop control)

```
void loop()
{
  unsigned long currentMillis = millis();
  unsigned long currentMicros;
  recvWithEndMarker();
  stepsToGo = computeStepsToGo();
  if (convertNewNumber())
  {
    Serial.print("Converted number: datanumber is: ");
    Serial.println(dataNumber);
    // Only get to this stage if there was new data to convert
    if (stepsToGo <= 0)
    {
      // Only get to this stage if not busy, otherwise will have thrown away input
      goToPosition(dataNumber);
      Serial.print("Got target position: ");
      Serial.println(targetPosition);


      /* Define number of steps in acceleration phase using Equation (3) */
      accelSteps = long(( maxPermissSpeed * maxPermissSpeed) / ( 2.0 * (double)maxAccel)); // Equation 4
      stepsToGo = computeStepsToGo();
      maxSpeed = maxPermissSpeed;

      if (2 * accelSteps > stepsToGo)
      {
        // Define maximum speed in profile and number of steps in acceleration phase
        maxSpeed = sqrt(minSpeed * minSpeed + stepsToGo * maxAccel); // Modified version of eq. 5
        accelSteps = (long)(stepsToGo / 2);
      }
      ps = ((double)ticksPerSec) / maxSpeed; // Eq 7

      p1 = (double)ticksPerSec / sqrt( minSpeed * minSpeed + 2 * maxAccel); // Eq 17 but need initial vel
      p = p1;
      R = (double) maxAccel / ((double)ticksPerSec * (double)ticksPerSec); // Eq 19
    }
  }
  /* Timed loop for stepping */
  currentMicros = micros();
  if (currentMicros - prevStepTime >= p)
  {
    moveOneStep();
    prevStepTime = currentMicros;
    computeNewSpeed();
  }
```

Runs only if we have a new position to move to!

Stepping loop!



$$S = (v^2 - v_0^2) / (2 \cdot a)$$ [4, 16],

$$v = (v_0^2 + 2 \cdot a \cdot S)^{1/2}$$ [5]

$$p_i = F / v_i$$ [7]

$$p_1 = F / (v_0^2 + 2 \cdot a)^{1/2}$$ [17],

$$R = a / F^2$$ [19].

Printing loop, slow!

```
  /* Timed loop for printing */
  if (currentMillis - prevMillisPrint >= printInterval)
  {
    // save the last time you printed output
    prevMillisPrint = currentMillis;
    printLoop();
  }
}
```

Experiment 3: Stepper Motor (Open-loop control)

```
/* Move a single step, holding pulse high for delayMicroSeconds */
void moveOneStep()
{
    if (p != 0) /* p=0 is code for "don't make steps" */
    {
        digitalWrite(stepPin, HIGH);
        if (direction == FWDS)
        {
            /* Is something missing here? */
            digitalWrite(dirPin, HIGH);
            currentPosition++;
        }
        else
        {
            /* Is something missing here? */
            digitalWrite(dirPin, LOW);
            currentPosition--;
        }
        delayMicroseconds(stepLengthMus);
        digitalWrite(stepPin, LOW);
    }
}
```

```
/* Calcuate new value of step interval p based on constants defined in loop() */
void computeNewSpeed()
{
    double q;
    double m;
    stepsToGo = computeStepsToGo();

    /* ----------------------------------------------------------------- */
    /* Start of on-the-fly step calculation code, executed once per step */
    if (stepsToGo == 0)
    {
        p = 0; // Not actually a zero step interval, used to switch stepping off
        return;
    }
    else if (stepsToGo > accelSteps && (long)p > long(ps))  //Speeding up
    {
        m = -R;  // definition following equation 20
    }
    else if (stepsToGo <= accelSteps)   //  Slowing down
    {
        m = R;
    }
    else   //  Running at constant speed
    {
        m = 0;
    }

    /* Update to step interval based on Eiderman's algorithm, using temporary variables */
    q = m * p * p; // this is a part of optional enhancement
    p = p * ( 1 + q + 1.5 * q * q); // this is an enhanced approximation -equation [22]
    /* Need to ensure rounding error does not cause drift outside acceptable interval range:
       replace p with relevant bound if it strays outside */
    if (p < ps)
    {
        p = ps;
    }
    if (p > p1)
    {
        p = p1;
    }
    /* End of on-the-fly step calculation code */
    /* ----------------------------------------------------------------- */
}
```

Summary

- Understand finer details of how a stepper motor is used
- Understand how to interface a stepper motor to a computer
- Appreciate the issues associated with generating the movements for a stepper motor
- Understand the stepper motor characteristics
- Link the lectures contents with what we will see in the lab next week!