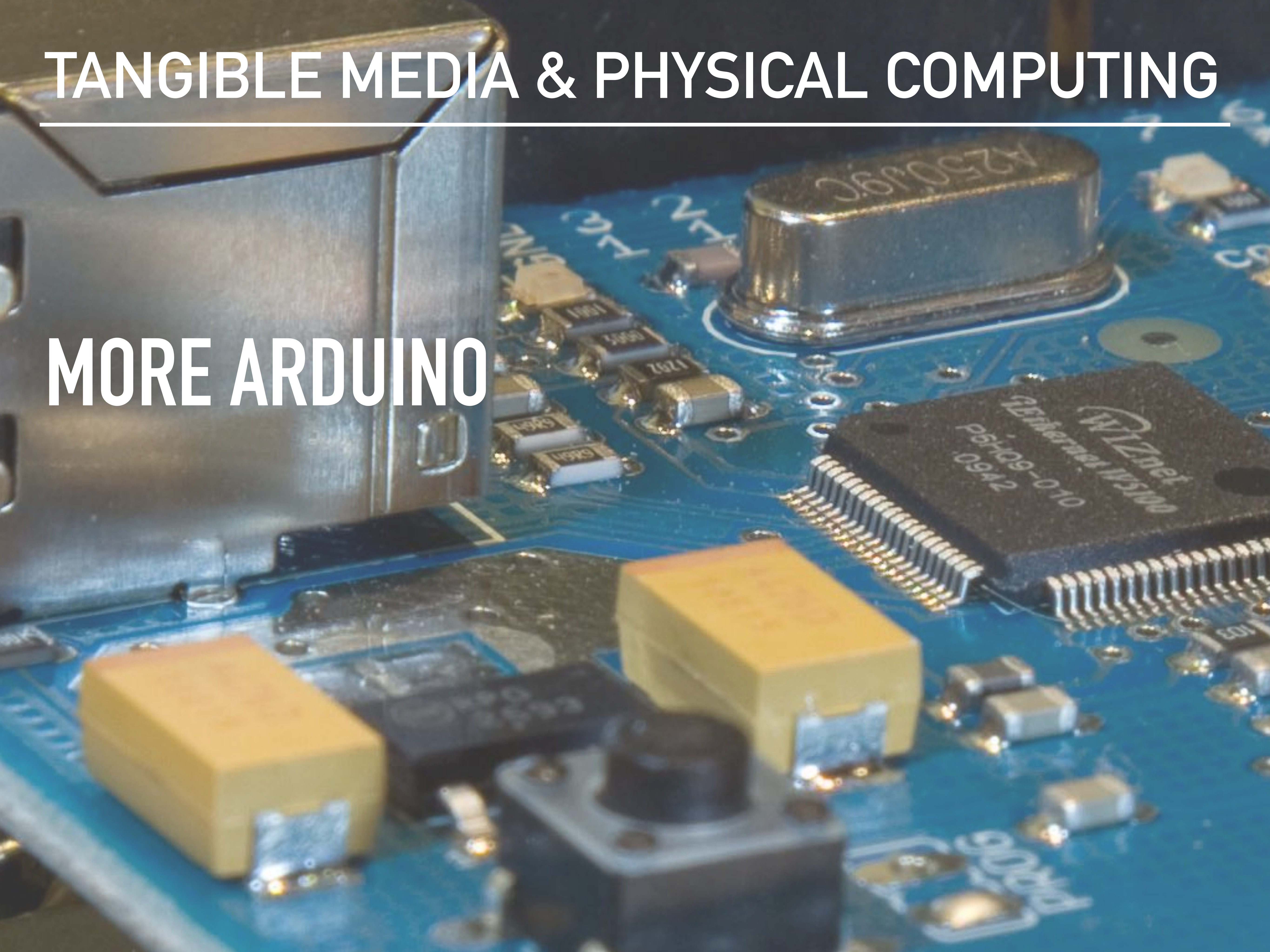
# TANGIBLE MEDIA & PHYSICAL COMPUTING

## MORE ARDUINO



# AGENDA

**RECAP**

ALGORITHMIC APPROACHES

TIMERS

# RECAP: LAST WEEK WE DID:

**ARDUINO IDE INTRO**
**MAKE SURE BOARD AND USB PORT SELECTED**
**UPLOAD PROCESS**
**COVERED DATATYPES**
**BASIC PROGRAMMING SYNTAX AND CONSTRUCTS**
**I/O : DIGITAL (R/W) AND ANALOG (R/W (PWM))**
**DELAY()**
**SERIAL DEBUGGING**

# VARIABLE RESISTORS & PULL UP/DOWN

**EXPLANATION ON BOARD**

# AGENDA

RECAP
**ALGORITHMIC APPROACHES**
TIMERS

# ALGORITHMIC APPROACHES

MICROCONTROLLERS CAN SENSE WHAT'S GOING ON IN THE PHYSICAL WORLD USING DIGITAL AND ANALOG SENSORS, BUT A SINGLE SENSOR READING DOESN'T TELL YOU MUCH. IN ORDER TO TELL WHEN SOMETHING SIGNIFICANT HAPPENS, YOU NEED TO KNOW WHEN THAT READING CHANGES.

WE WILL LOOK AT HOW  TO DETECT FOR THREE COMMON CHANGES IN SENSOR READINGS THAT GIVE YOU INFORMATION ABOUT REAL WORLD EVENTS:
STATE CHANGE DETECTION ON DIGITAL SENSORS, AND THRESHOLD CROSSING AND PEAK DETECTION ON ANALOG SENSORS. YOU'LL USE THESE THREE TECHNIQUES ALL THE TIME WHEN YOU'RE DESIGNING TO READ USERS' ACTIONS.

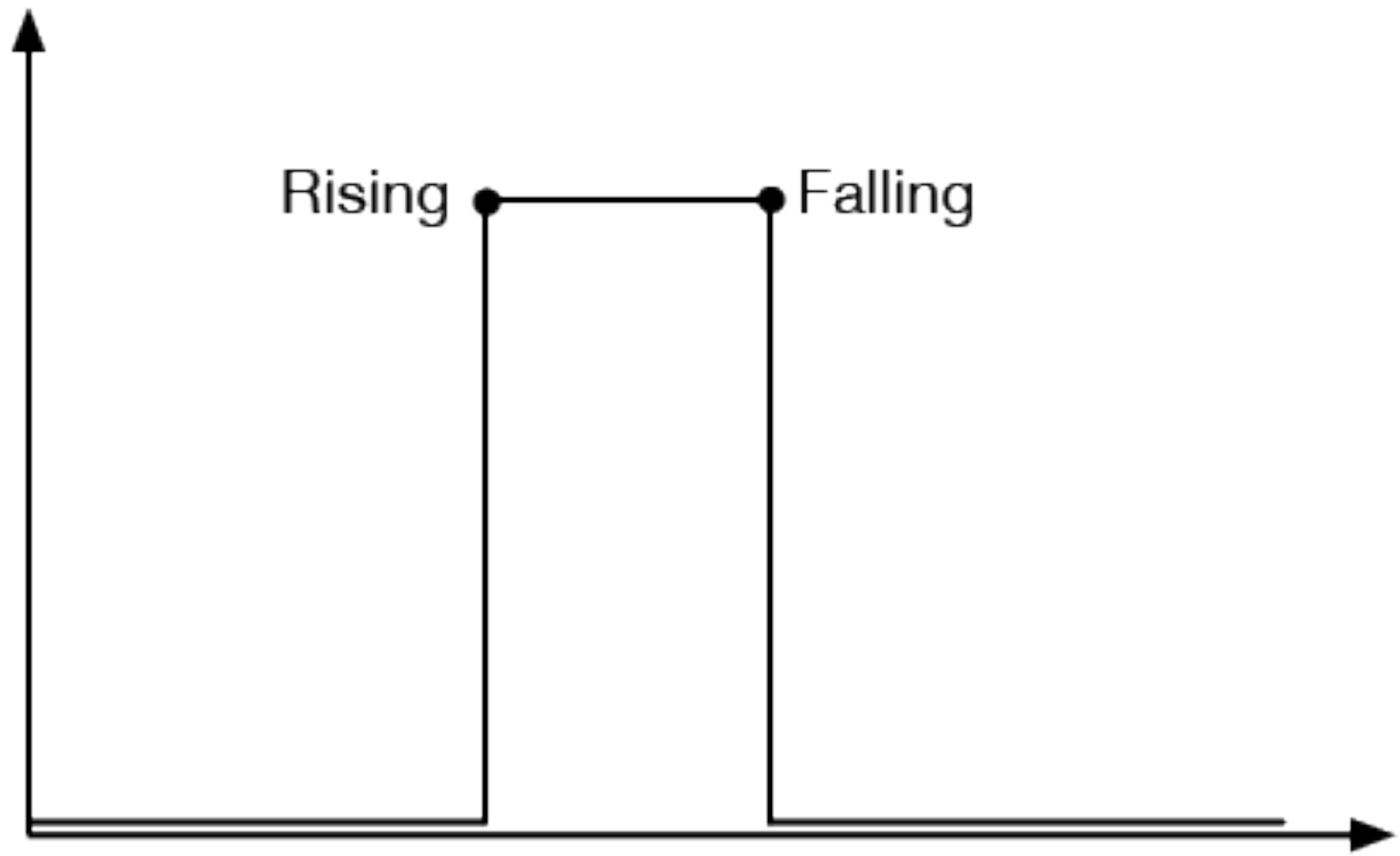
# SENSOR CHANGES

SENSOR CHANGES ARE DESCRIBED IN TERMS OF THE CHANGE IN VOLTAGE OUTPUT OVER TIME.
THE MOST IMPORTANT CASES TO CONSIDER FOR SENSOR CHANGE ARE :

THE RISING AND FALLING EDGES OF A DIGITAL OR BINARY SENSOR,
THE RISING AND FALLING EDGES AND THE PEAK OF AN ANALOG SENSOR.

THE FOLLOWING GRAPHS OF SENSOR VOLTAGE OVER TIME ILLUSTRATE THESE CONDITIONS:

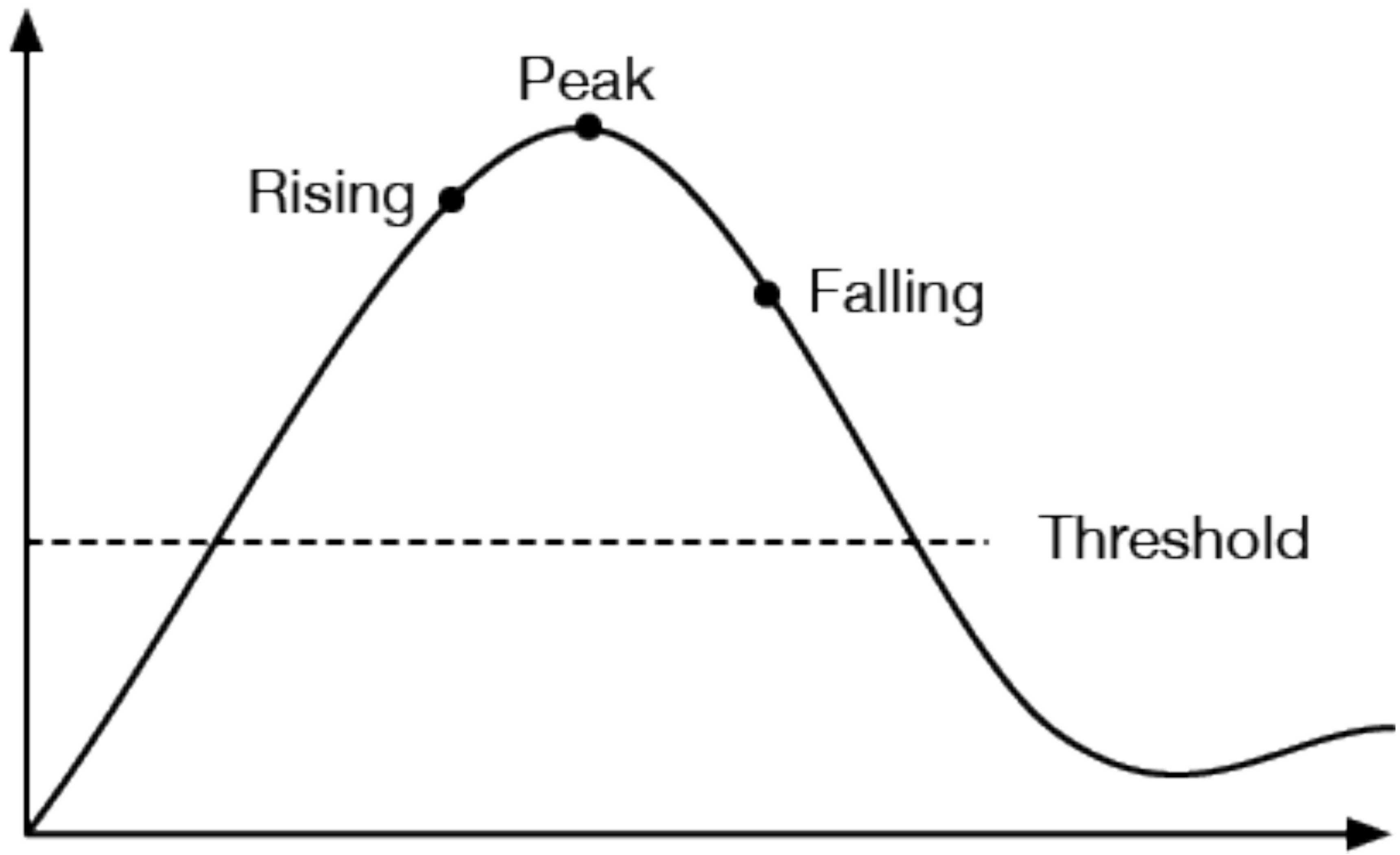
# SENSOR CHANGES : DIGITAL




**DIGITAL SENSORS CHANGE FROM HIGH VOLTAGE TO LOW AND VICE VERSA.**

**THE CHANGE FROM LOW VOLTAGE TO HIGH IS CALLED THE RISING EDGE, & THE CHANGE FROM HIGH VOLTAGE TO LOW IS CALLED THE FALLING EDGE.**

# SENSOR CHANGES: ANALOG




**THE THREE GENERAL STATES OF AN ANALOG SENSOR ARE :**

**RISING (CURRENT STATE > PREVIOUS STATE),**
**WHEN IT'S FALLING (CURRENT STATE < PREVIOUS STATE),**
**AND WHEN IT'S AT A PEAK.**

# SENSOR CHANGES : DIGITAL

**TO TELL THAT A DIGITAL SENSOR IS CURRENTLY ACTIVE (I.E. A BUTTON IS PRESSED) WHEN WIRED WITH A PULL DOWN RESISTOR, WE CAN FORMULATE THE FOLLOWING EXPRESSION**

```
if (digitalRead(BUTTON_PIN == HIGH)
{
    // we know that the sensor has been
   activated...
}
```

**HOWEVER - WE WANT TO KNOW SOMETHING MORE ....**

# DIGITAL STATE CHANGE DETECTION

## DID THE DIGITAL SENSOR JUST CHANGE?
## NEED A VARIABLE TO HOLD THE PREVIOUS BUTTON STATE:

```
int prevButtonState = LOW;    // global var

void loop() {
  int buttonState = digitalRead(BUTTON_PIN);

if (buttonState != prevButtonState) {
      // do stuff if it is different here
  }

  // save button state for next comparison:
  prevButtonState = buttonState;
}
```

# APPLICATION: COUNTING PRESSES

```
int prevButtonState = LOW;
int buttonPresses = 0;  // #of button presses

void loop() {
  int buttonState = digitalRead(BUTTON_PIN);

  if (buttonState != prevButtonState) {
      // do stuff if it is different here

      if (buttonState == HIGH) {
          buttonPresses++
      }
  }
  prevButtonState = buttonState;
}
```

# SENSOR CHANGES : ANALOG

WHEN YOU'RE USING ANALOG SENSORS, BINARY STATE CHANGE DETECTION IS NOT USUALLY EFFECTIVE, BECAUSE YOUR SENSORS CAN HAVE MULTIPLE STATES (1024).

THE SIMPLEST FORM OF ANALOG STATE CHANGE DETECTION IS TO LOOK FOR THE SENSOR TO RISE ABOVE A GIVEN THRESHOLD IN ORDER TO TAKE ACTION.

IF YOU WANT THE ACTION BE TRIGGERED ONLY ONCE WHEN YOUR SENSOR PASSES THE THRESHOLD, YOU NEED TO KEEP TRACK OF BOTH ITS CURRENT STATE AND PREVIOUS STATE.

# SENSOR CHANGES : ANALOG SIMPLE

```
int threshold = 512;
 // an arbitrary threshold value

void loop() {
   // read the sensor:
   int sensorVal = analogRead(A0);

   // if it's above the threshold:
   if (sensorVal >= threshold) {
       //do something
   }

}
```

# SENSOR CHANGES : ANALOG: RISING

```
int prevSenseState = 0; int threshold = 512;

void loop() {
  int sensorVal = analogRead(A0);

  if (sensorVal >= threshold) {

    // Was previous Val was BELOW threshold?

    if(prevSenseState < threshold){
      // now we do something ONCE
    }
  }
  prevSenseState = sensorVal;
}
```

# SENSOR CHANGES : ANALOG: FALLING

```
int prevSenseState = 0; int threshold = 512;

void loop() {
  int sensorVal = analogRead(A0);

  if (sensorVal <= threshold) {

    // Was previous Val was ABOVE threshold?

    if(prevSenseState > threshold){
      // now we do something ONCE
     }
  }
  prevSenseState = sensorVal;
}
```

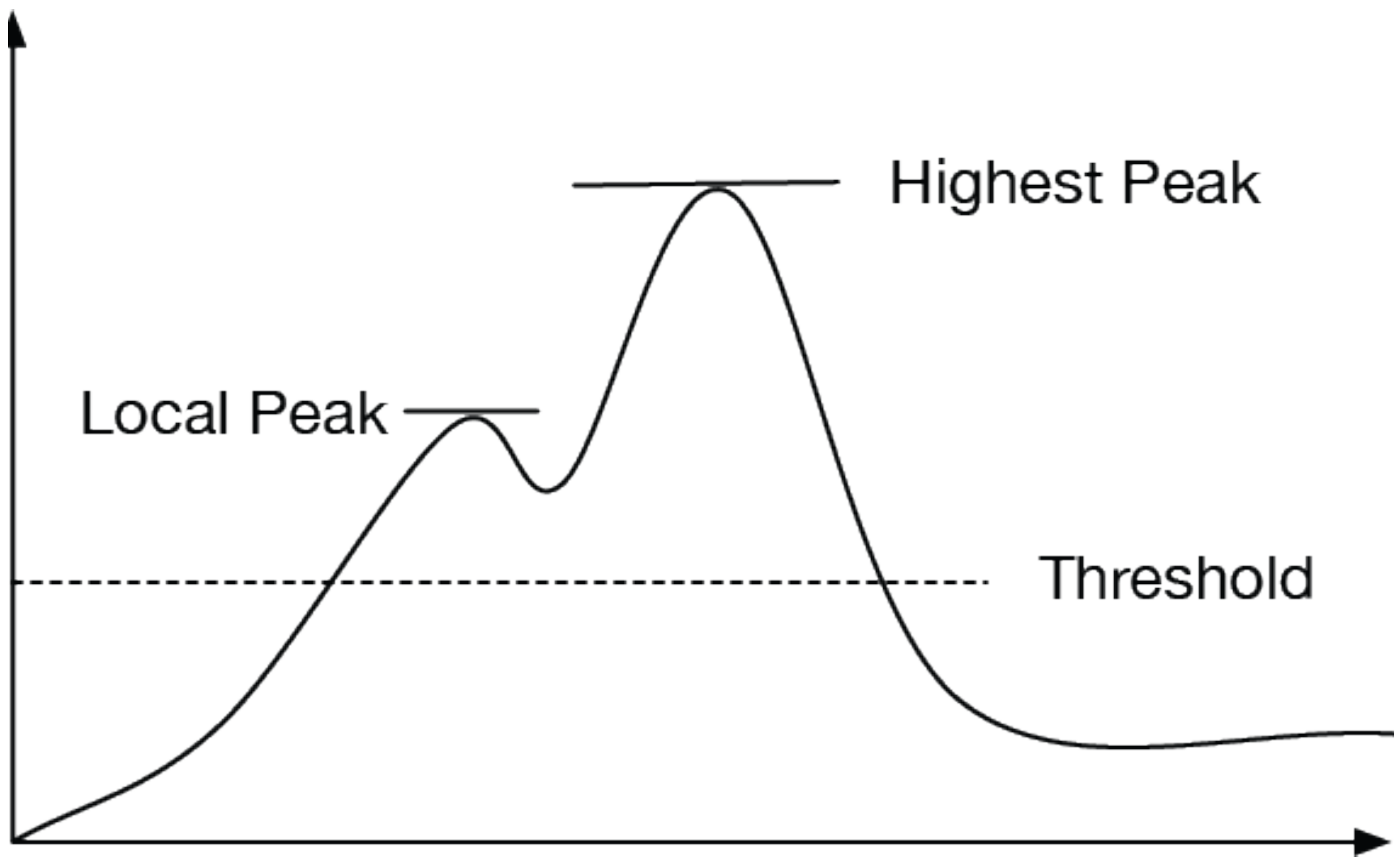
# PEAK DETECTION

```
int peakValue = 0;

 void loop() {
   //read sensor on pin A0:
   int sensorValue = analogRead(A0);

   // check if it's higher than the current peak:
   if (sensorValue > peakValue) {
      // set a new peak
      peakValue = sensorValue;
   }
 }
```

# PEAK DETECTION

```
int peakValue = 0; int threshold = 50;

void loop() {
  int sensorValue = analogRead(A0);
  if (sensorValue > peakValue) {
    peakValue = sensorValue;
  }

  if (sensorValue <= threshold) {

      if (peakValue > threshold){
      //Have a peak value: do something
     // & reset peak variable:
     peakValue = 0;
    }
  }
}
```

# DEALING WITH NOISE




QUITE OFTEN, YOU GET NOISE FROM SENSOR READINGS THAT CAN INTERFERE WITH PEAK READINGS.
INSTEAD OF A SIMPLE CURVE, YOU GET A JAGGED RISING EDGE FILLED WITH MANY LOCAL PEAKS:

# DEALING WITH NOISE

```
int peakValue = 0; int threshold = 50; int noise=5;

void loop() {
  int sensorValue = analogRead(A0);
  if (sensorValue > peakValue) {
    peakValue = sensorValue;
  }

  if (sensorValue <= threshold - noise) {

      if (peakValue > threshold + noise){
     //Have a peak value: do something
     // & reset peak variable:
     peakValue = 0;
    }
  }
}
```

# LET'S FILTER NOISY DATA …

MEASUREMENTS FROM THE REAL WORLD OFTEN CONTAIN NOISE.

NOISE IS JUST THE PART OF THE SIGNAL YOU DIDN'T WANT & FILTERING IS A METHOD TO REMOVE SOME OF THE UNWANTED SIGNAL TO LEAVE A SMOOTHER RESULT.

# AVERAGE FILTER

ONE OF THE EASIEST WAYS TO FILTER NOISY DATA IS BY AVERAGING:

ADD TOGETHER A NUMBER OF MEASUREMENTS, THEN DIVIDE THE TOTAL BY THE NUMBER OF MEASUREMENTS YOU ADDED TOGETHER.

THE MORE MEASUREMENTS YOU INCLUDE IN THE AVERAGE, THE MORE NOISE GETS REMOVED.

# AVERAGE FILTER FUNCTION

```
int calcAverage() {

        float sumSenseVal = 0;
        int samplesToAverage = 16; // arbitrary

        for(int i = 0; i < samplesToAverage; i++){
             averageSenseVal+=readSenseVal();
             delay(1);
        }

        int averageVal =
              sumSenseVal / samplesToAverage;

        return averageVal;
}
```

# RUNNING AVERAGE FILTER FUNCTION

ONE DISADVANTAGE OF THE AVERAGE FILTER IS THE AMOUNT OF TIME NEEDED TO MAKE A MEASUREMENT.

AN ALTERNATIVE TO TAKING ALL THE MEASUREMENTS AT ONCE, THEN AVERAGING THEM IS:
TO TAKE ONE MEASUREMENT AT A TIME AND ADD IT TO A RUNNING AVERAGE.

# RUNNING AVERAGE FILTER FUNCTION

```
const int RUNNING_SAMPLES= 16;
int runningAverageBuffer[RUNNING_SAMPLES];
int nextCount =0;

void loop(){
    int rawSenseVal = analogRead(SENSE_PIN);
    runningAverageBuffer[nextCount] = rawSenseVal;
    nextCount++;
    if (nextCount >= RUNNING_SAMPLES)
        nextCount = 0;

    int currentSum= 0;
    for(int i=0; i< RUNNING_SAMPLES; i++){
        currentSum+= runningAverageBuffer[i];
    }
    int averageVal = currentSum / RUNNING_SAMPLES;
    delay(100);
}
```

# WEIGHTED AVERAGE FILTER FUNCTION

## THE LAST APPROACH TO FILTER AN ANALOG SENSOR READING IS BY TAKING A WEIGHTED AVERAGE OF SAMPLES OF THE SENSOR.

## IT'S BASED ON THIS ALGORITHM:

```
filteredValue = weight*rawValue+(1-weight)*lastFilteredValue
```

WEIGHT IS A VALUE BETWEEN 0 AND 1 THAT INDICATES HOW RELIABLE THE NEW RAW VALUE IS.

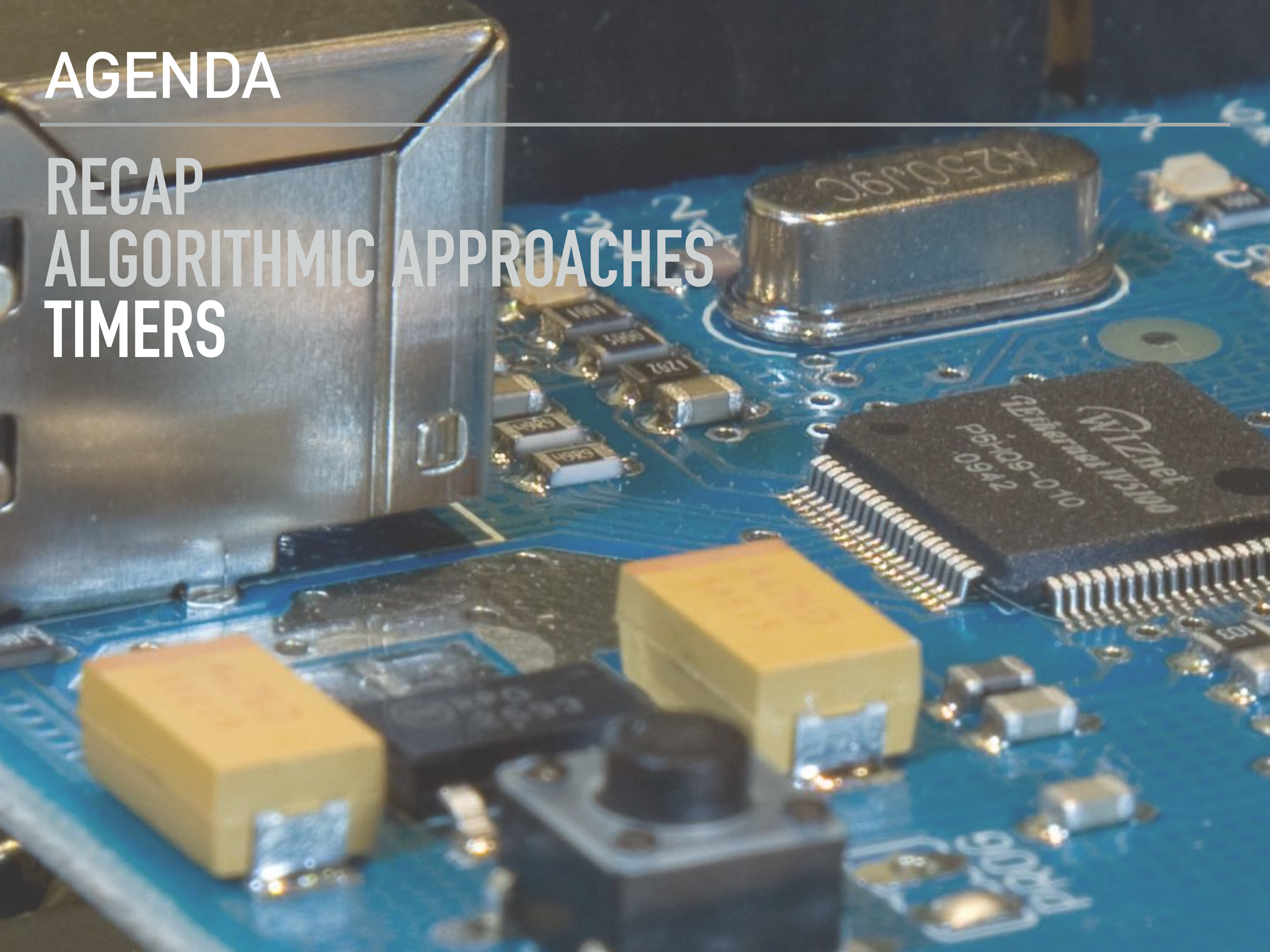
IF IT'S 100% RELIABLE, WEIGHT = 1, AND NO FILTERING IS DONE.
IF IT'S TOTALLY UNRELIABLE, WEIGHT = 0, AND THE RAW RESULT IS FILTERED OUT.

# WEIGHTED AVERAGE FILTER FUNCTION

```
const float weight = 0.5;
float prevEst = 0.0;

void loop() {
        int sensorVal = analogRead(A0);

       // filter the sensor's result:
       float currEst= filter(sensorVal, weight, prevEst);

       // save the current result for future use:
       prevEst= currentEst;
}

// filter the current result using a weighted avg filter:
float filter (float rawValue, float w, float lastValue) {
      float result = w * rawValue + (1.0-w)*lastValue;
      return result;
}
```

# AGENDA

## RECAP
## ALGORITHMIC APPROACHES
## TIMERS



# TIMING FUNCTIONS

TIMING IS VERY IMPORTANT – ELECTRONICS ARE NOT INSTANTANEOUS AND MOST COMPONENTS REQUIRE SOME TIME BEFORE THEY CAN BE ACCESSED. QUERYING THE SENSOR BEFORE IT IS READY CAN RESULT IN MALFORMED DATA OR RETRIEVING A PREVIOUS RESULT.

IN THIS INSTANCE ONE NEEDS TO USE DELAY():

TELLS THE MICRO CONTROLLER TO WAIT TO THE SPECIFIED NUMBER OF MS BEFORE RESUMING THE SKETCH

# TIMING FUNCTIONS

**DELAYMICROSECONDS():**
SIMILAR TO DELAY() BUT INSTEAD OF WAITING MS – THE PARAMETER TO THE FUNCTION IS IN MICROSECONDS.

**MILLIS() :**
RETURNS THE NUMBER OF MS THAT THE SKETCH HAS BEEN RUNNING FOR . CAN ALSO BE USED EFFECTIVELY TO CALCULATE HOW LONG SOME ACTION TAKES…

**MICROS():**
SIMILAR TO MILLIS() EXCEPT IT COUNTS IN MICROSECONDS.

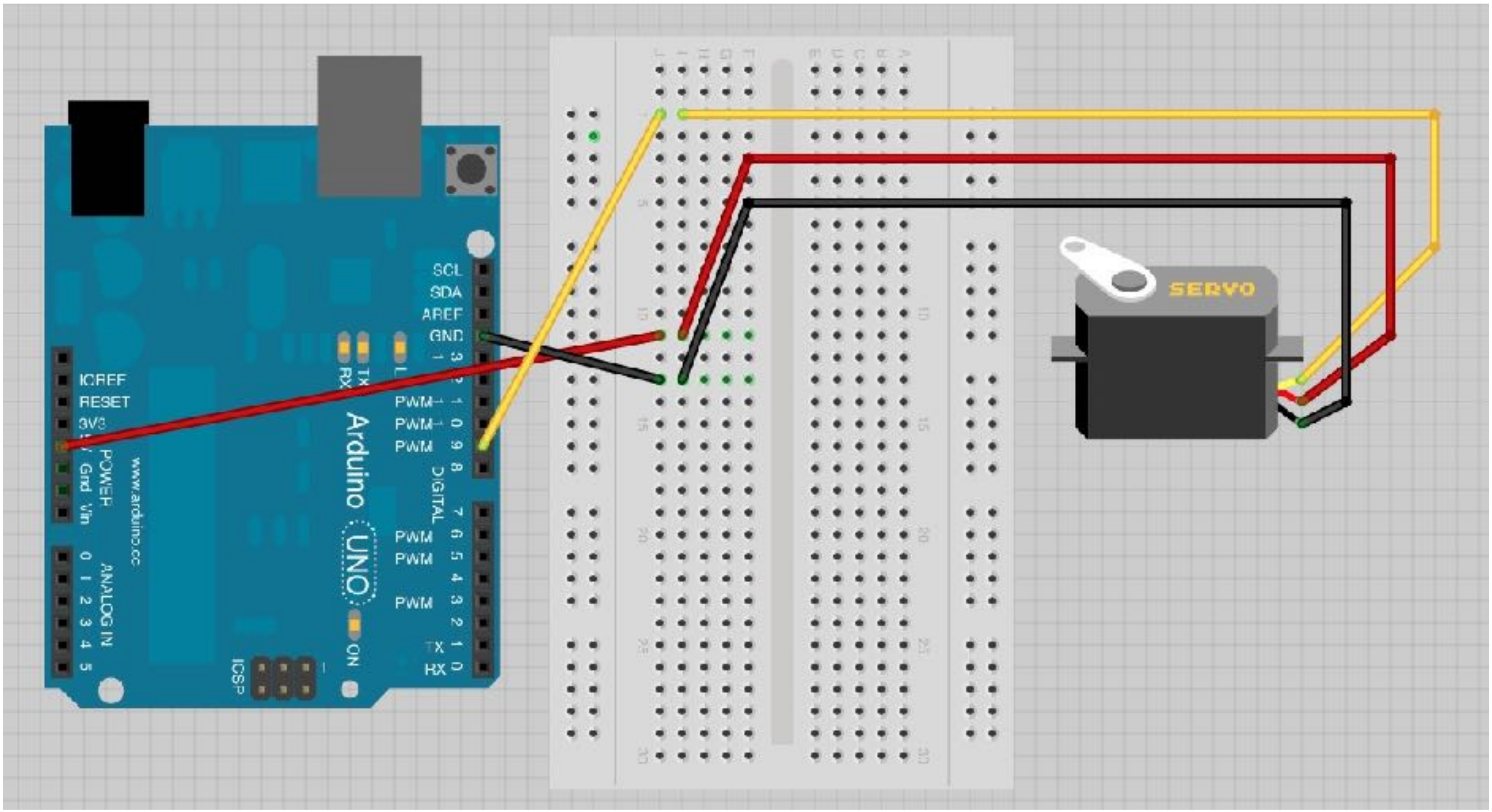
# TIMING CONCEPTS : BACK TO BLINK

## BUILD THE CIRCUIT ON YOUR BREADBOARD
## ADD THE CODE AND UPLOAD

```
#define LED_PIN 13

void setup() {
pinMode(LED_PIN, OUTPUT);
}

void loop() {
digitalWrite(LED_PIN, HIGH);
delay(1000);                    //wait for a second
digitalWrite(LED_PIN, LOW);
delay(1000);                    // wait for a second
}
```

# TIMING CONCEPTS: A SIMPLE MOTOR

## LETS DO ANOTHER ARDUINO EXAMPLE USING A SERVO MOTOR: (CODE IS CALLED "A") USE SAME BREADBOARD.



# TIMING CONCEPTS: MOTOR & LED BLINK

WHAT HAPPENS IF WE TRY TO BLINK AND SWEEP?
(TAKE THE OTHER EXAMPLE FROM SLACK CALLED "B")

SWEEP USES THE DELAY() TO CONTROL THE SWEEP SPEED.
BLINK USES THE DELAY() BETWEEN TURNING OFF/ON LED

WHEN COMBINING THE BASIC BLINK SKETCH WITH THE SERVO SWEEP EXAMPLE, YOU WILL FIND THAT IT ALTERNATES BETWEEN BLINKING AND SWEEPING. BUT IT WON'T DO BOTH SIMULTANEOUSLY.

# TIMING CONCEPTS: TIMERS

LETS BUILD A TIMER!

WE WILL USE A SIMPLE TECHNIQUE FOR IMPLEMENTING TIMING BY KEEPING TRACK OF A TIMER WHICH RECORDS HOW MUCH TIME HAS PASSED SINCE THE TIMER STARTED.

INSTEAD OF USING A DELAY() - WE WILL JUST CHECK REGULARLY OUR TIMER - TO SEE IF IT IS TIME FOR AN ACTION TO BE TAKEN

MEANWHILE THE PROCESSOR IS FREE TO DO OTHER THINGS….

# TIMING CONCEPTS: BLINK 1

LET'S CHANGE OUR BLINK SKETCH TO USE A TIMER INSTEAD OF A DELAY()… OPEN THE APPROPRIATE CODE FROM SLACK: C SKETCH
UPLOAD & TEST – IS IT DIFFERENT?

FUNCTIONALLY:  NO
BUT IT ILLUSTRATES AN IMPORTANT CONCEPT:
A STATE MACHINE

THE PROGRAM REMEMBERS THE CURRENT STATE OF THE LED AND THE LAST TIME IT CHANGED

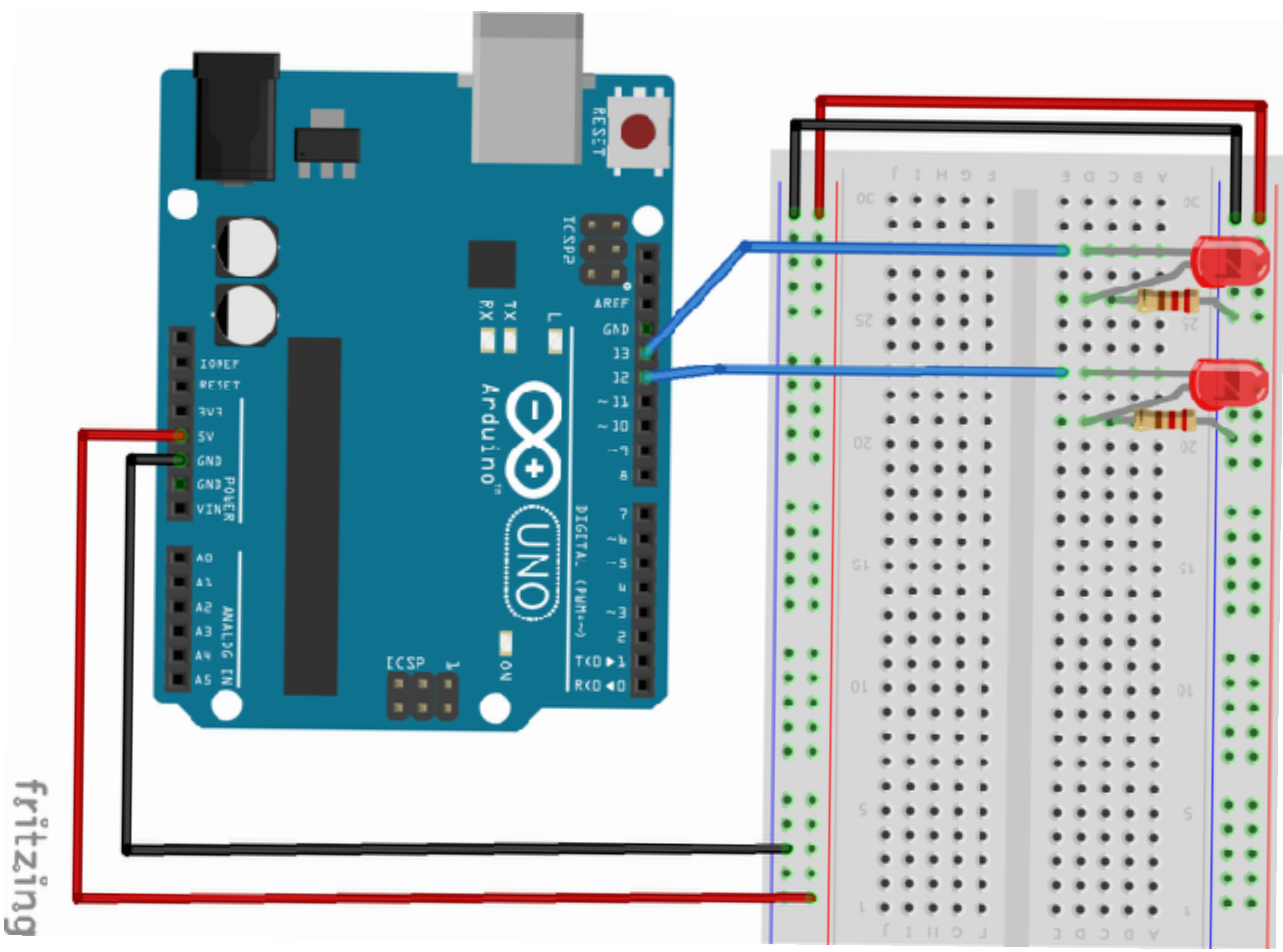
# TIMING CONCEPTS: BLINK 2

**LETS MAKE A MODIFIED SKETCH – WHERE NOW WE WANT TO FLASH THE LED WITH A DIFFERENT ON AND OFF TIME:**
**– WITHOUT A DELAY –**

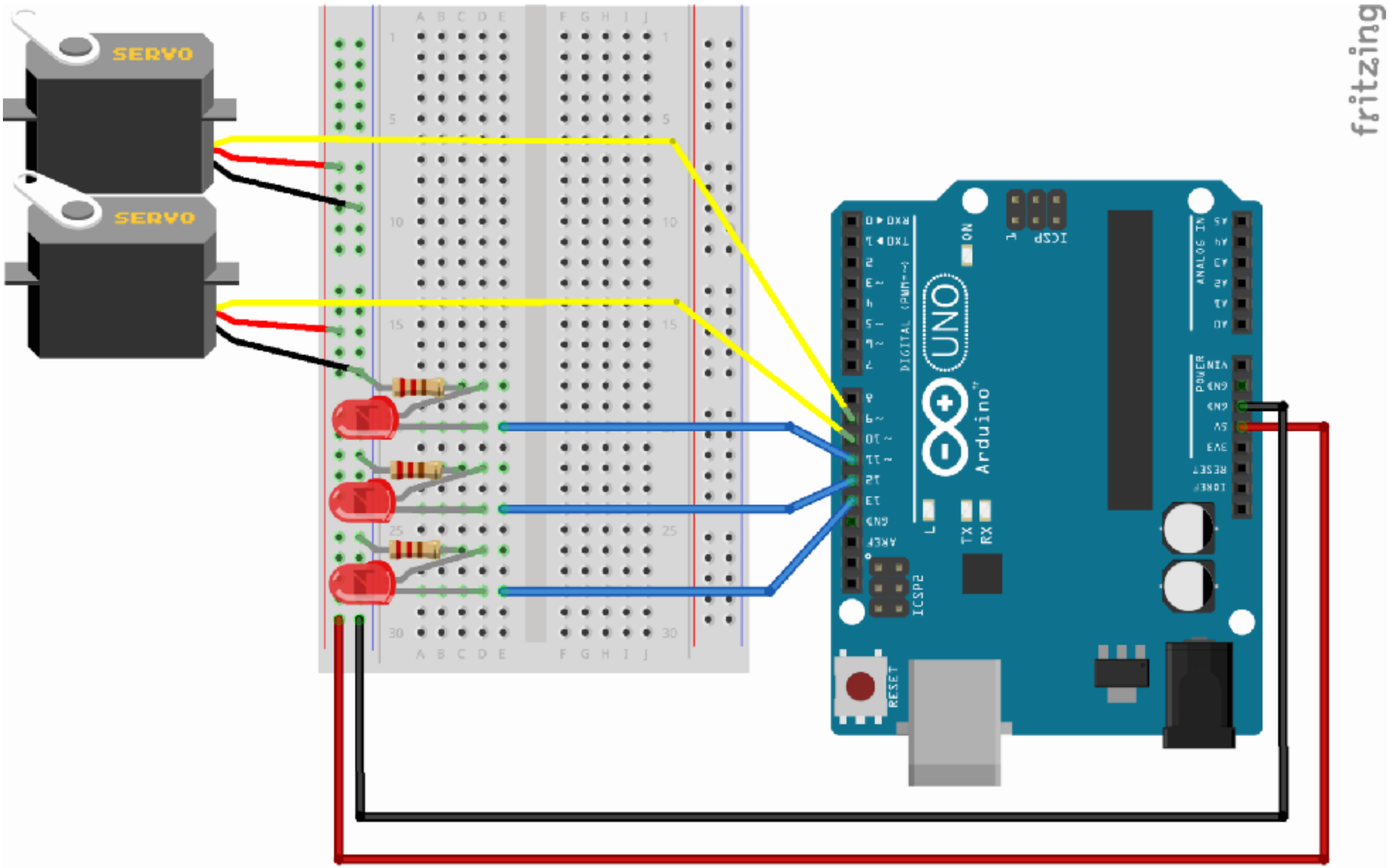
**CODE: “D” (IN SLACK)**

# TIMING CONCEPTS: 2 LEDS

## OK – NOW WE ARE GOING TO TRY AND HAVE TWO LEDS EACH BLINKING AT DIFFERENT RATES: (E)



# TIMING CONCEPTS: 2 MOTORS + 3 LEDS



# TIMING CONCEPTS: ADD IN A BUTTON