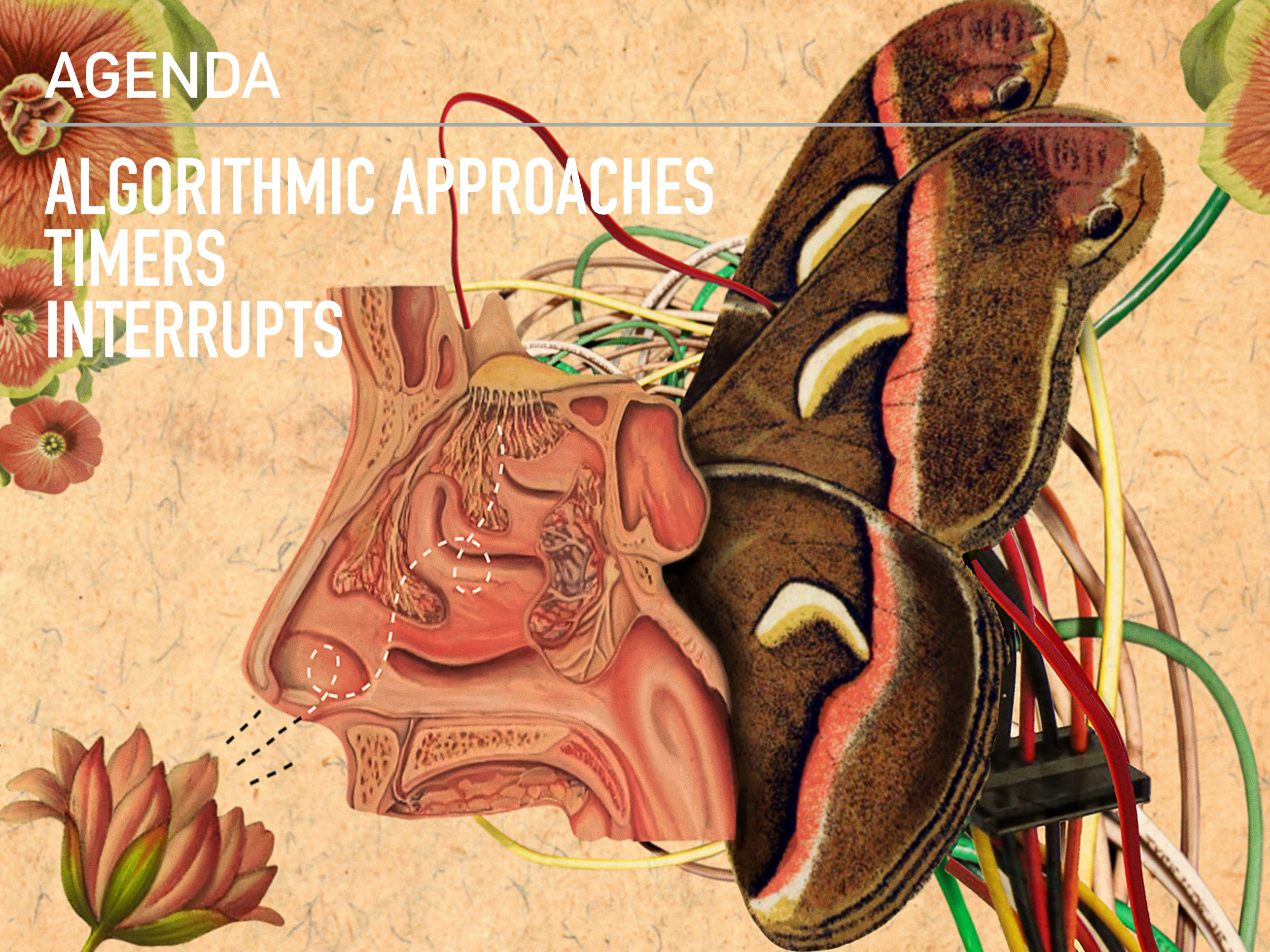
TANGIBLE MEDIA & PHYSICAL COMPUTING

ALGORITHMIC APPROACHES

AGENDA

ALGORITHMIC APPROACHES
TIMERS
INTERRUPTS



ALGORITHMIC APPROACHES

MICROCONTROLLERS CAN SENSE THE PHYSICAL WORLD USING DIGITAL AND ANALOG SENSORS. ALTHOUGH, A SINGLE SENSOR READING DOESN'T TELL YOU MUCH. IN ORDER TO DETERMINE WHEN SOMETHING SIGNIFICANT HAPPENS, YOU NEED TO KNOW WHEN THAT DATA POINT HAS CHANGED.

ALGORITHMIC APPROACHES

WE WILL CONSIDER HOW TO OBSERVE AND DETECT CHANGE OVER EITHER A DIGITAL OR ANALOG INPUT. IN GENERAL, THERE ARE THREE COMMON OBSERVATIONS OF SENSOR READINGS – THESE INFORM US OF THE REAL WORLD EVENTS, AND YOU'LL USE THESE THREE TECHNIQUES ALL THE TIME.

STATE CHANGE

THRESHOLD CROSSING

PEAK DETECTION

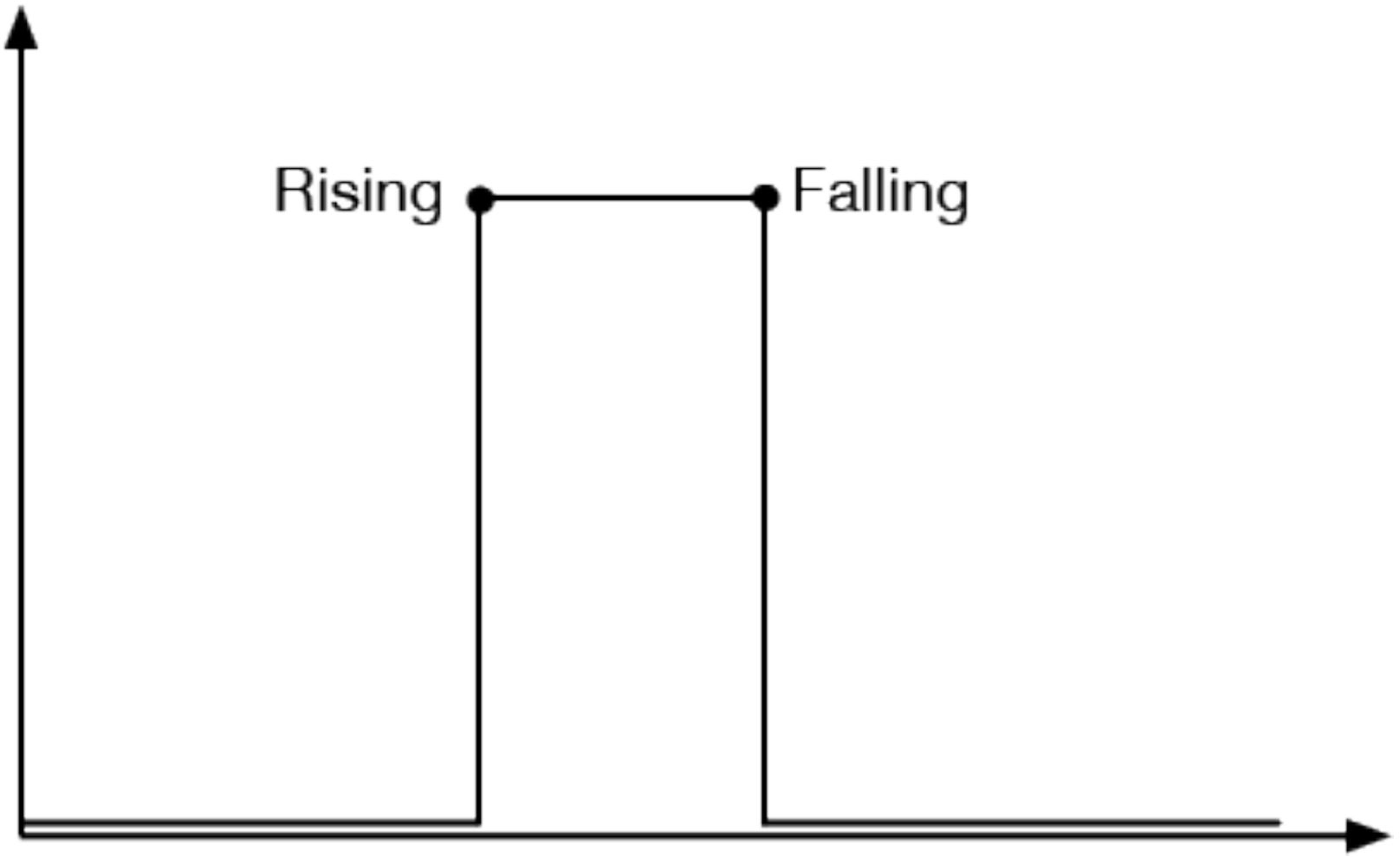
SENSOR CHANGES

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

THE RISING AND FALLING EDGES OF A DIGITAL OR BINARY SENSOR.

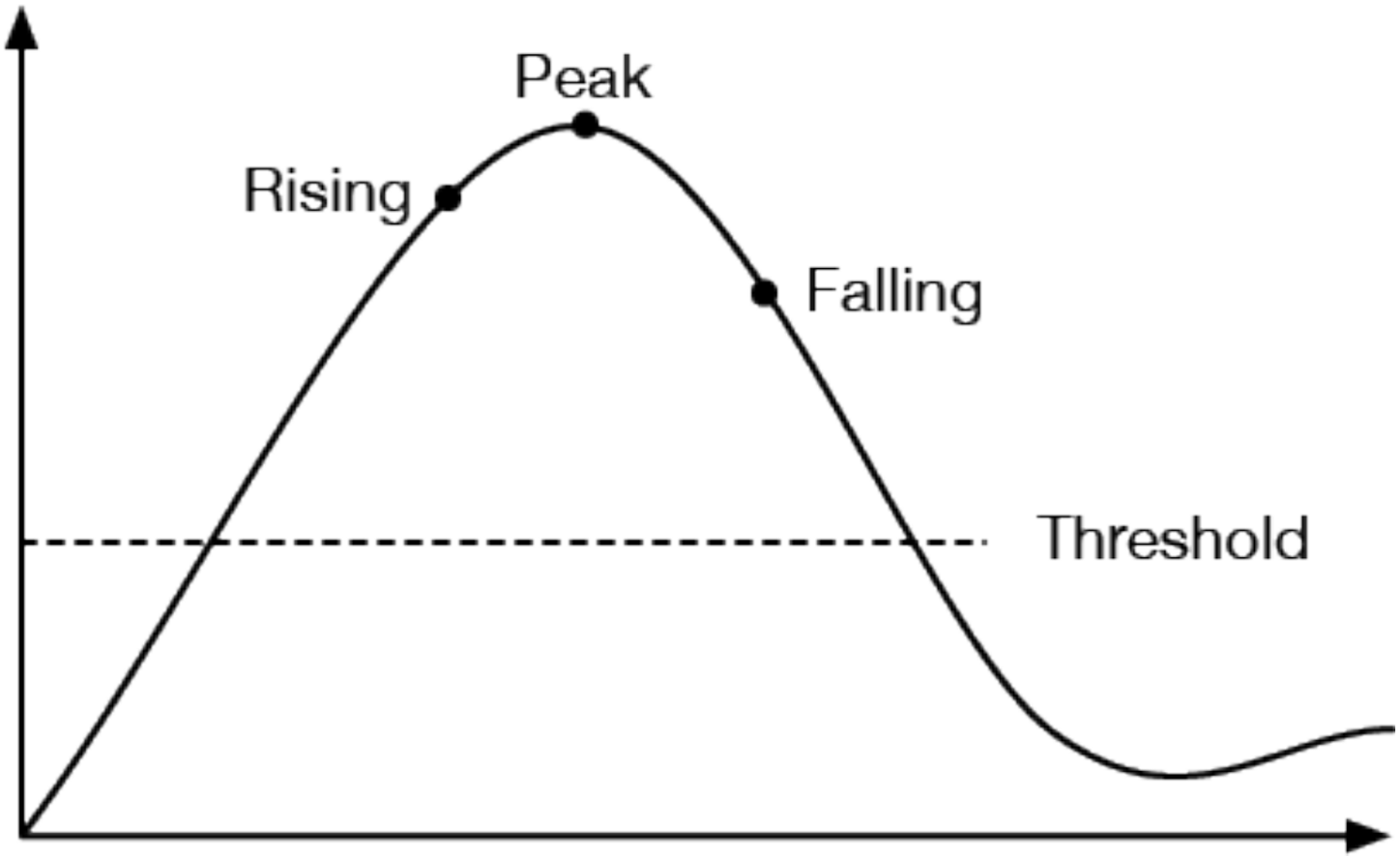
THE RISING, FALLING EDGES AND THE PEAK OF AN ANALOG SENSOR.

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 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 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

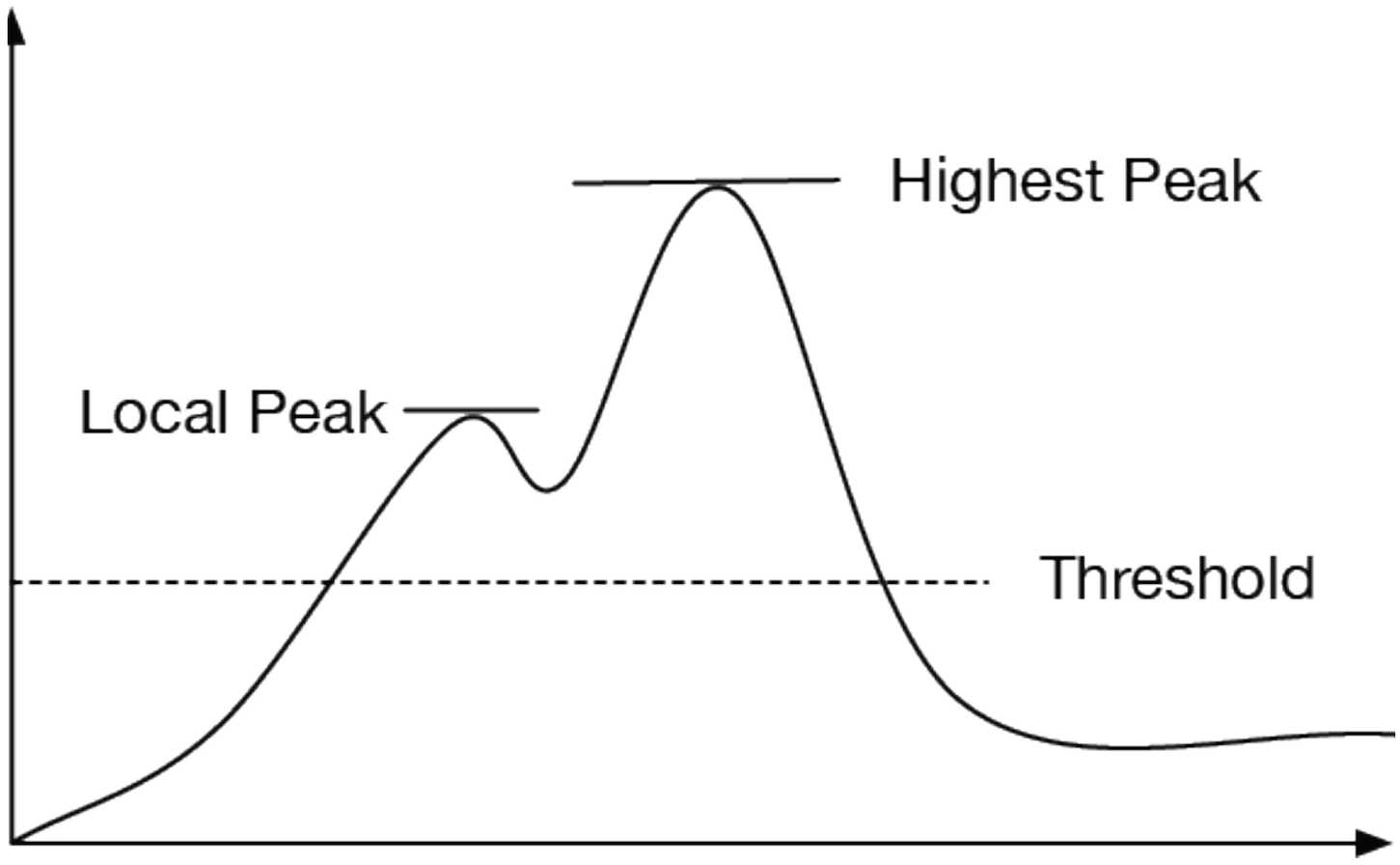
```
int peakValue = 0; int threshold = 50;

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 (SMOOTHED).

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

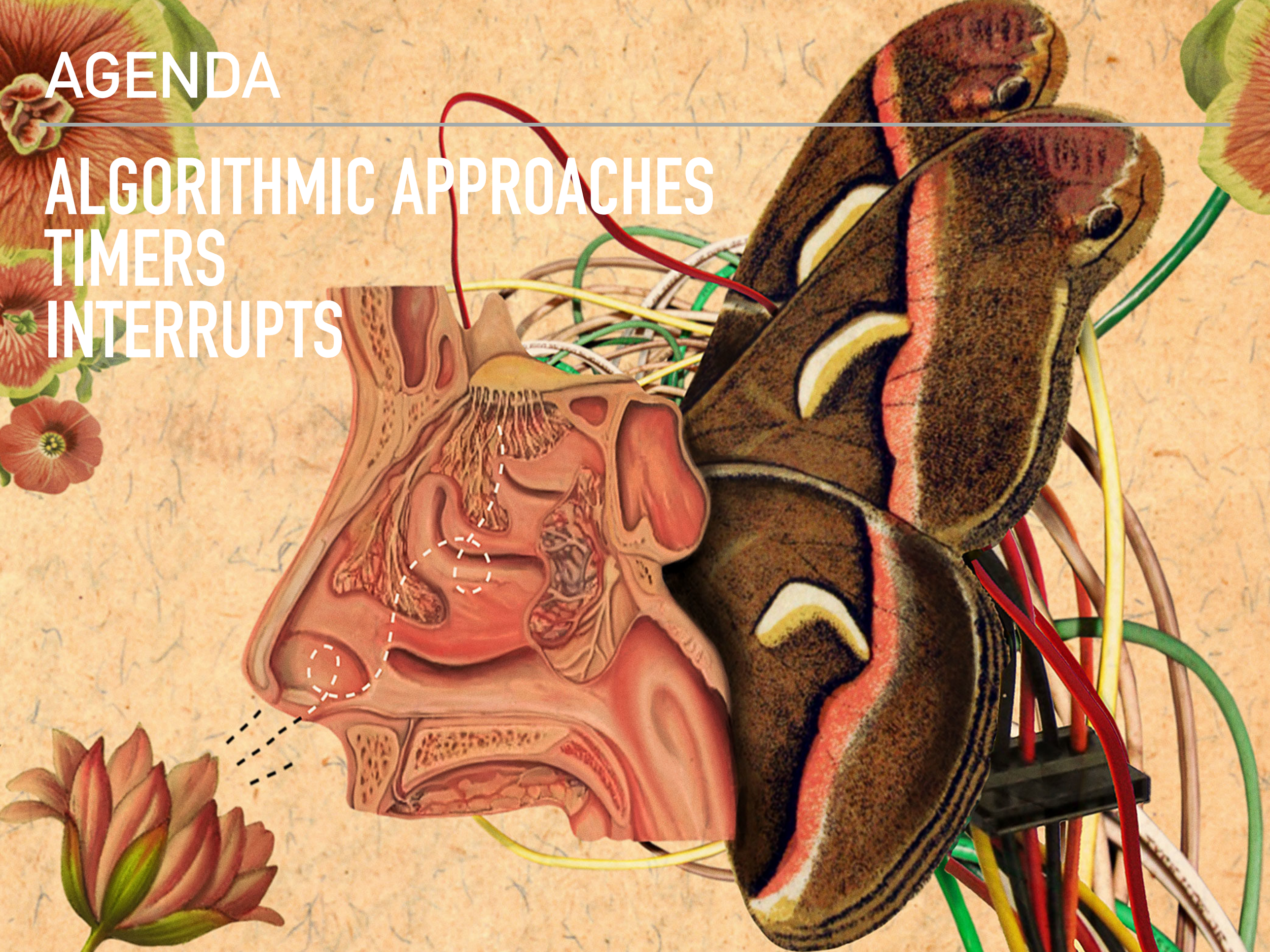
```
const int RUNNING_SAMPLES= 16;
int runningAverageBuffer[RUNNING_SAMPLES];
int nextCount =0;

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);
}
```

AGENDA

ALGORITHMIC APPROACHES
TIMERS
INTERRUPTS



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.

DELAY()

(JUST PLAIN EVIL *)

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: TIMERS

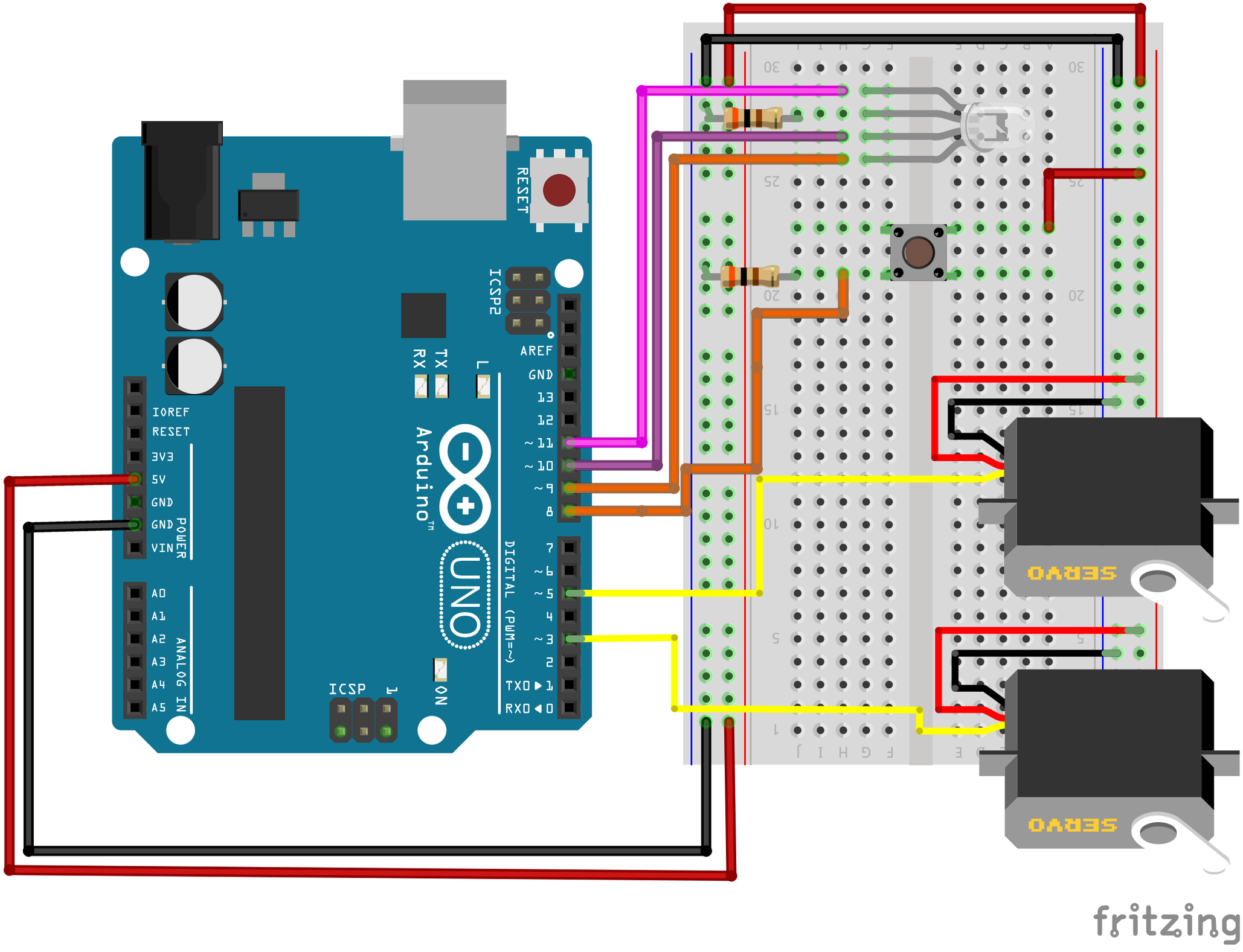
LETS BUILD A TIMER (BUT NOT TODAY)!

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 OUR TIMER REGULARLY – TO SEE IF IT IS TIME FOR AN ACTION TO BE TAKEN

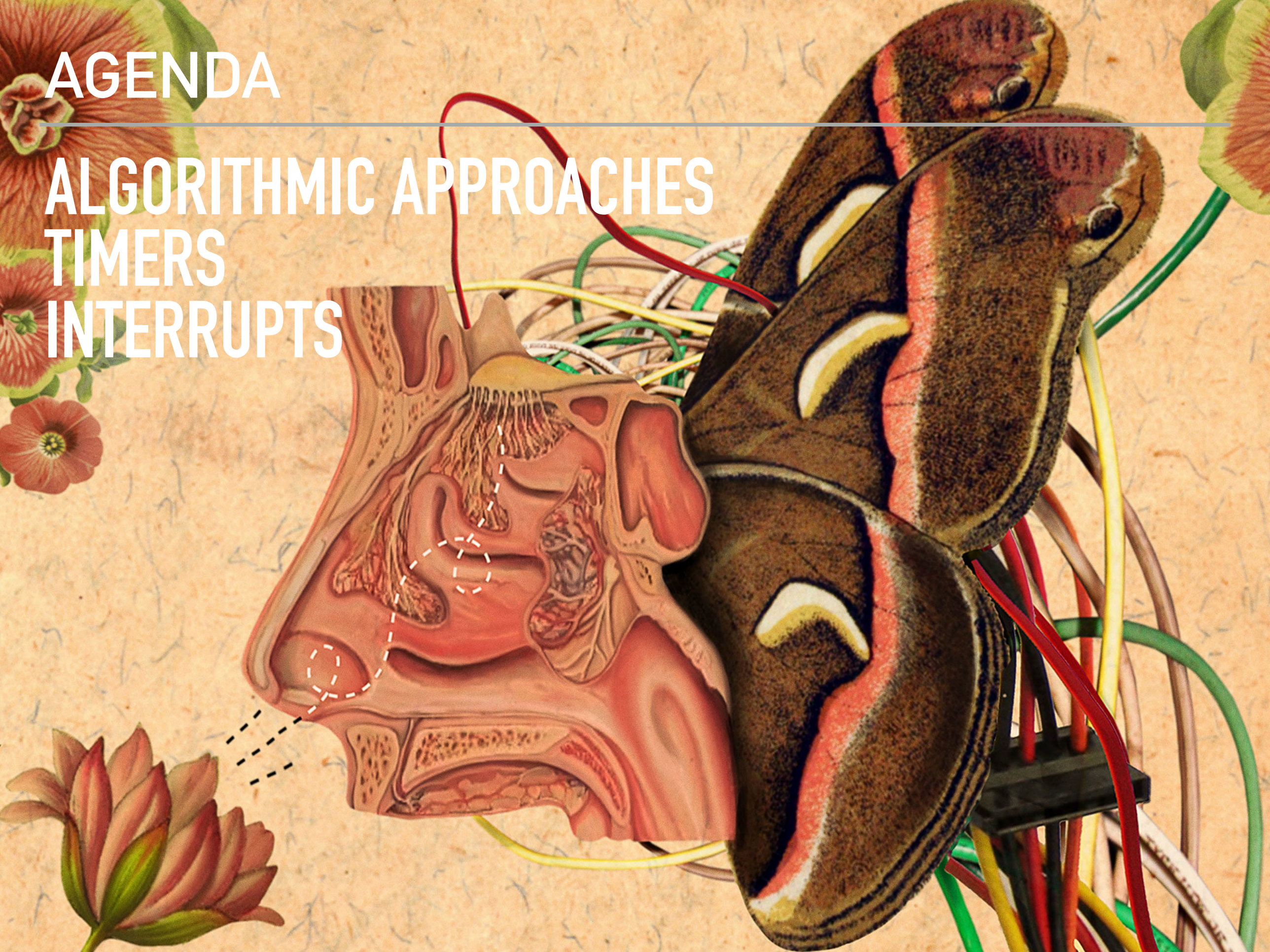
MEANWHILE THE PROCESSOR IS FREE TO DO OTHER THINGS. . . .

TIMING CONCEPTS: TIMED STATE



AGENDA

ALGORITHMIC APPROACHES
TIMERS
INTERRUPTS



INTERRUPTS

INTERRUPTS ARE USEFUL FOR MAKING THINGS HAPPEN AUTOMATICALLY IN MICROCONTROLLER PROGRAMS AND CAN HELP SOLVE* TIMING PROBLEMS. GOOD TASKS FOR USING AN INTERRUPT MAY INCLUDE READING A ROTARY ENCODER OR IMMEDIATELY ADDRESSING USER INPUT.

(LOVED EVILNESS *)

INTERRUPTS

INTERRUPTS ALLOW CERTAIN IMPORTANT TASKS TO HAPPEN IN THE BACKGROUND. SOME FUNCTIONS WILL NOT WORK WHILE INTERRUPTED (DELAY() & MILLIS()), AND INCOMING COMMUNICATION MAY BE IGNORED.

INTERRUPTS CAN SLIGHTLY DISRUPT THE TIMING OF CODE, HOWEVER, AND MAY BE DISABLED FOR PARTICULARLY CRITICAL SECTIONS OF CODE.

INTERRUPT SERVICE ROUTINE

ISRS ARE SPECIAL KINDS OF FUNCTIONS THAT HAVE SOME UNIQUE LIMITATIONS THAT MOST OTHER FUNCTIONS DO NOT. AN ISR CANNOT HAVE ANY PARAMETERS AND THEY DO NOT RETURN ANYTHING. AN ISR SHOULD BE AS SHORT AND FAST AS POSSIBLE. ISRS ARE PRIORITY DRIVEN, ONLY ONE CAN RUN AT A TIME, OTHER INTERRUPTS WILL BE EXECUTED THEREAFTER.

INTERRUPT SERVICE ROUTINE

```
// http://gammon.com.au/interrupts

/* INTERUPT DIRECTIVES: RISING, FALLING, CHANGE or LOW
attachInterrupt(digitalPinToInterrupt(pin), ISR, mode) */

const byte ledPin = 13;
const byte interruptPin = 2;
volatile byte state = LOW;
```

```
void setup() {
 pinMode(ledPin, OUTPUT);
 pinMode(interruptPin, INPUT_PULLUP);
 attachInterrupt(digitalPinToInterrupt(interruptPin), blink, CHANGE);
}

void loop() {
 digitalWrite(ledPin, state);
}

void blink() {
 state = !state;
}
```