ENG20009
Engineering Technology Inquiry Project

Unit Convenor : Dr Rifai Chai

Email: rchai@swin.edu.au

Phone: 9214 8119

Office: EN606B

Swinburne University of Technology

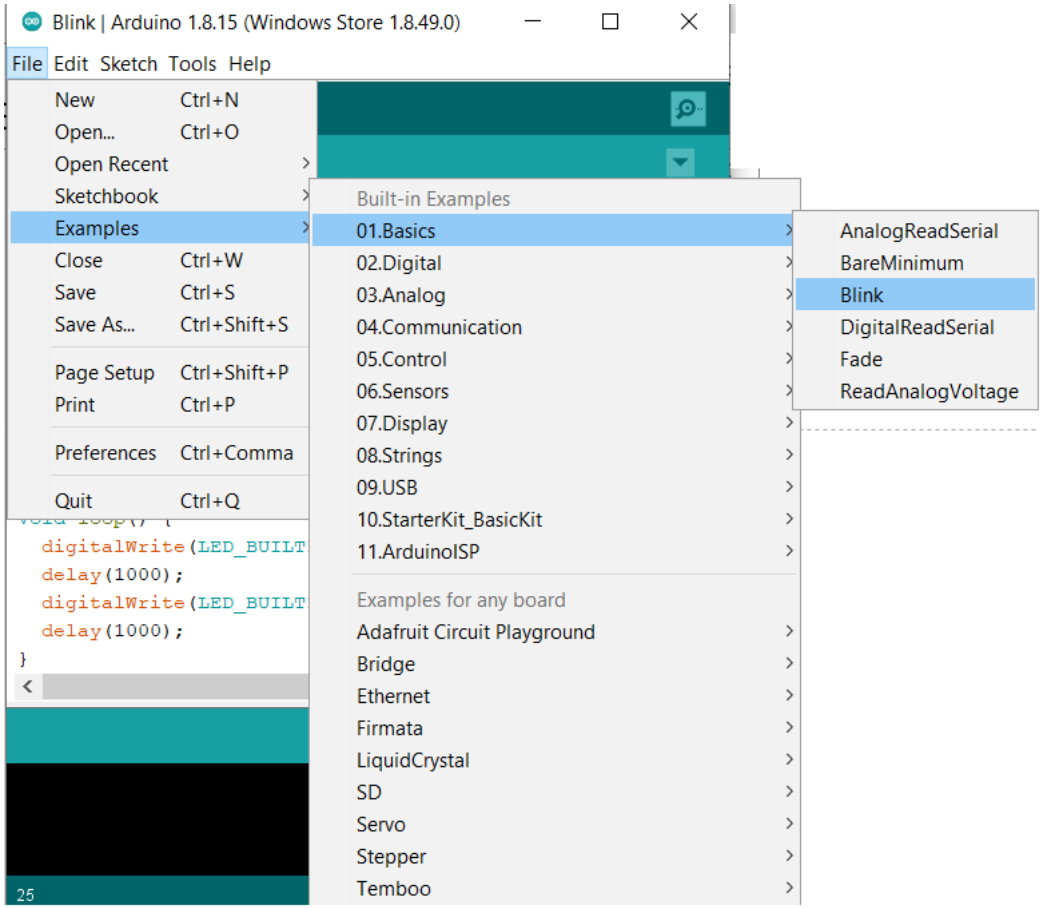
Seminar 2 - GPIO

Topics:

1. Hardware testing
2. GPIO – Outputs
3. GPIO – PWM
4. LED Bar
5. LED Matrix
6. Button
7. Button with no resistor
8. Button with Debouncing

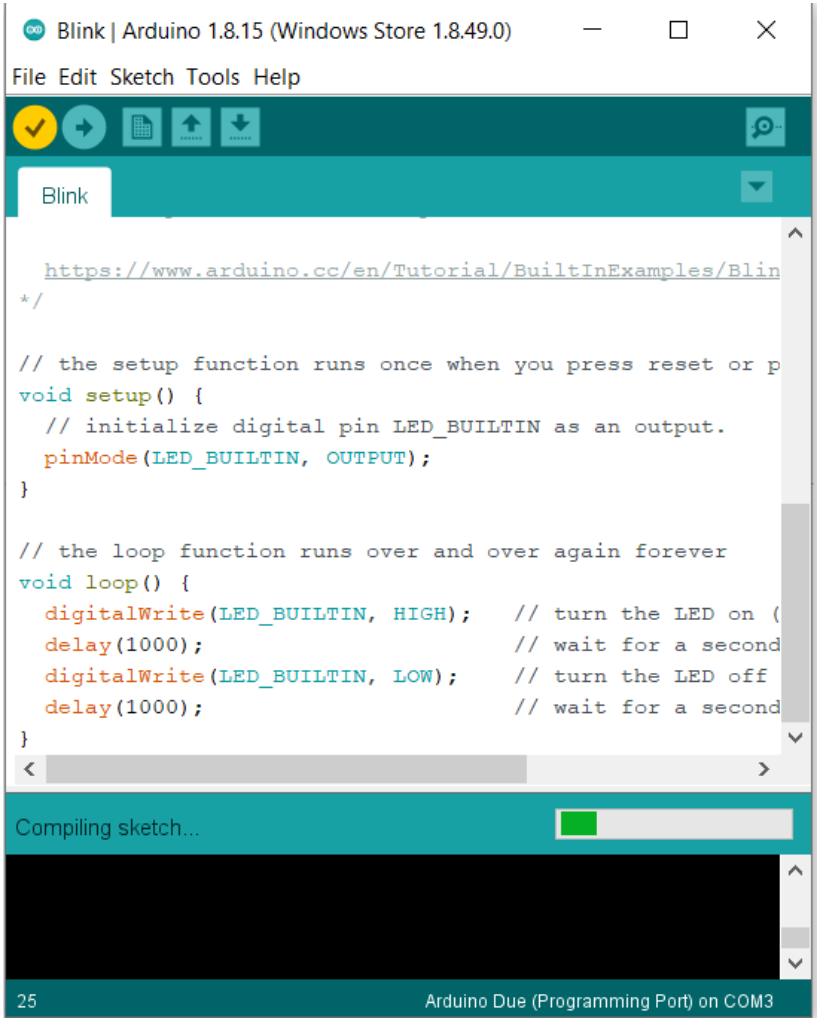
1. Hardware – Testing

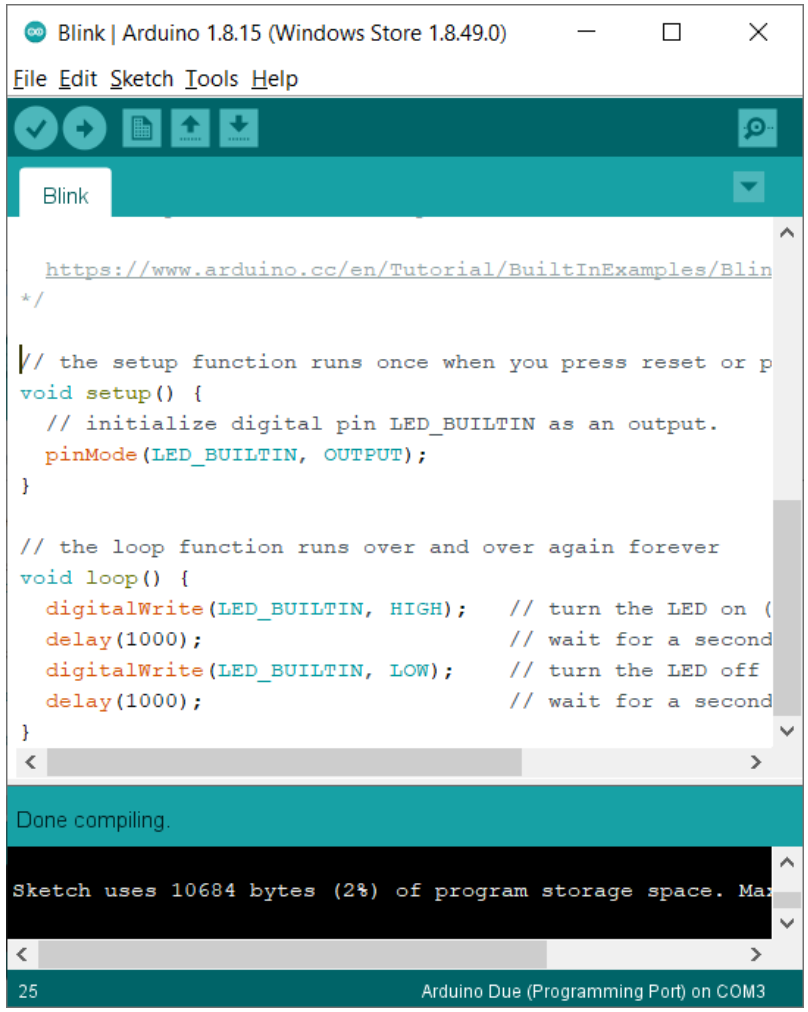
- ❑ The initial testing can be done by test to program the internal LED
- ❑ Example ‘Blink’ program can be used.



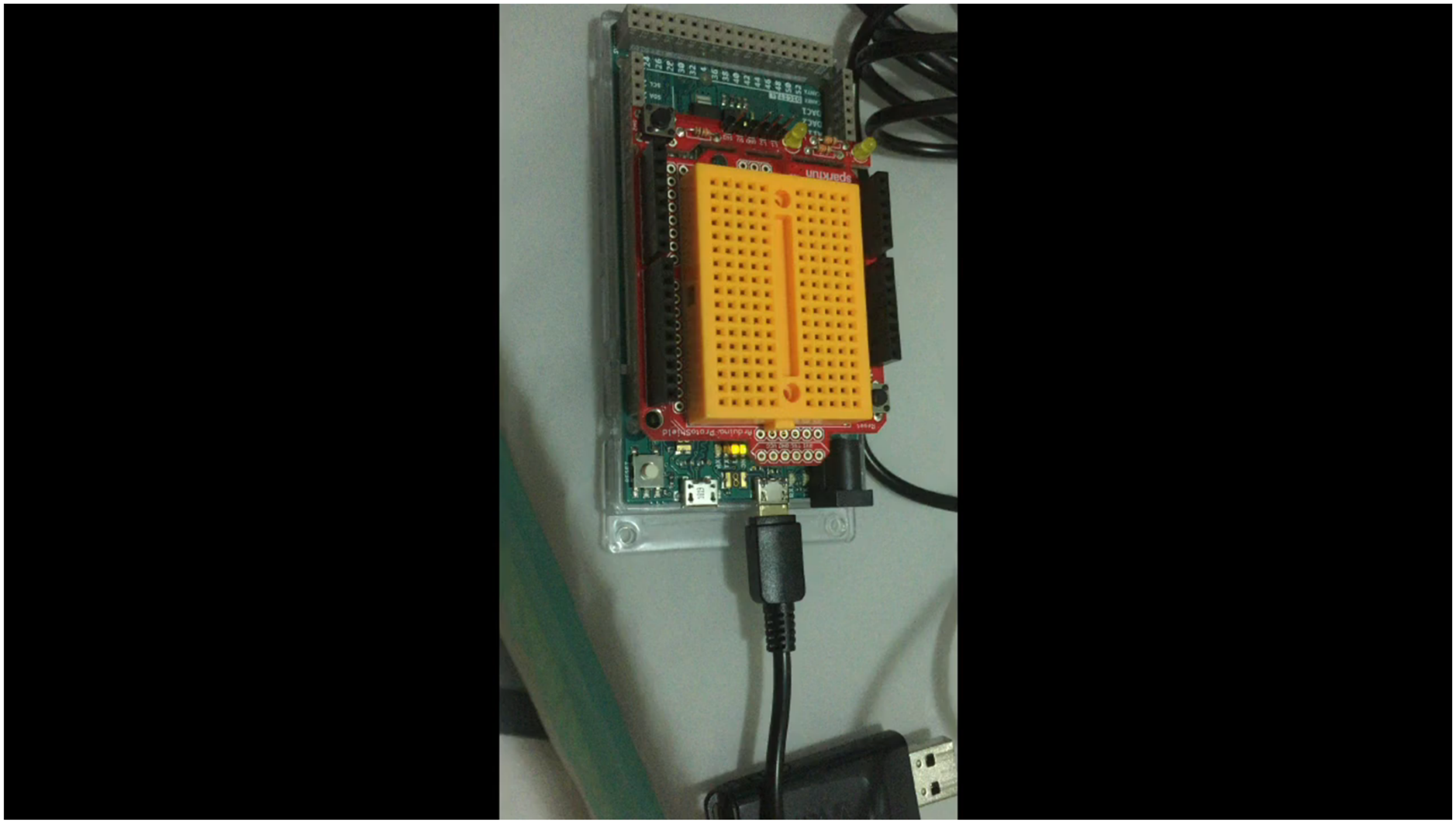
Hardware – Testing

- Compiling the code



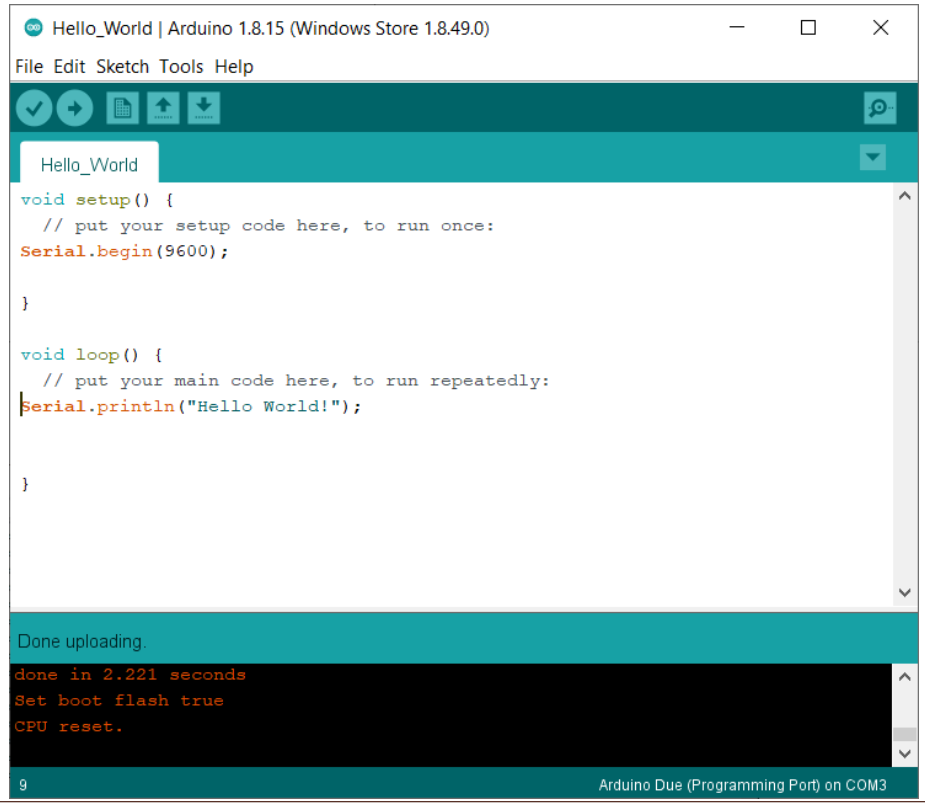


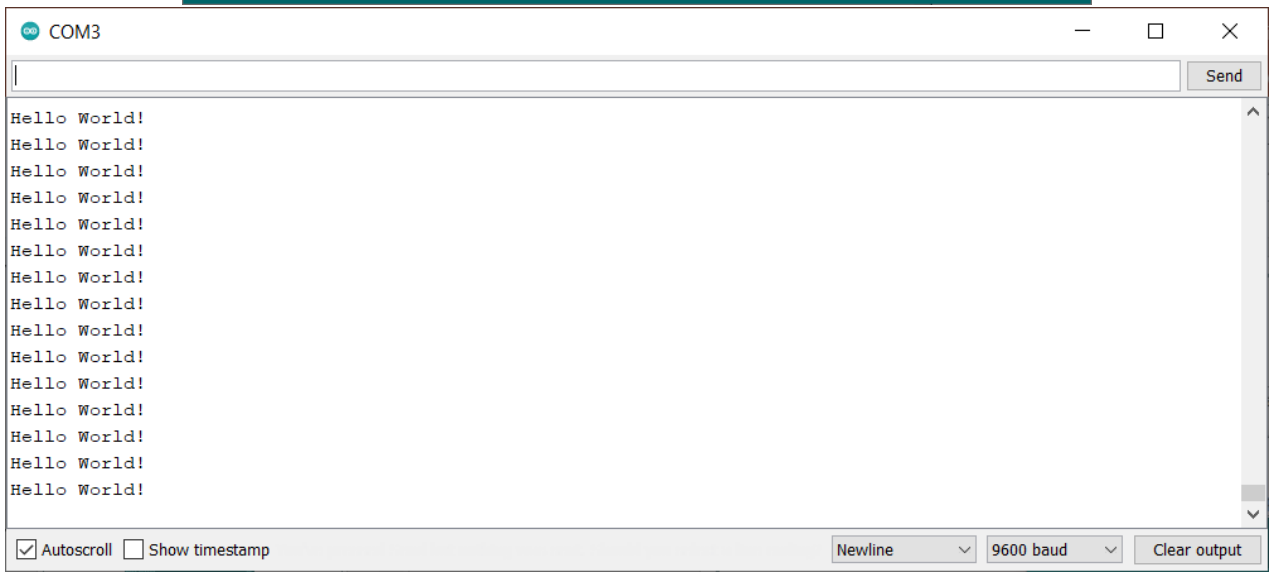
Hardware – Arduino Due – Testing



Hardware – Arduino Due – Hello World

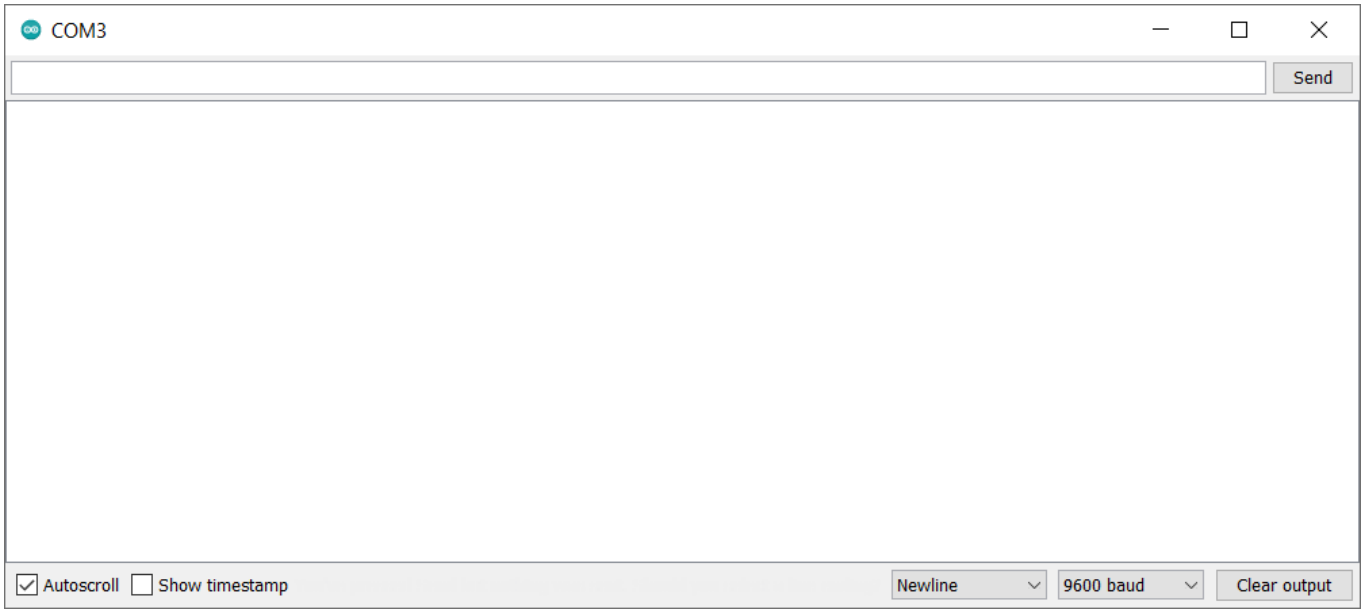
Hardware – Arduino Due – Hello World





Displaying Data from the Arduino in the Serial Monitor

- To open the Serial Monitor, start the IDE and click the Serial Monitor icon button on the tool bar, shown in below.
- Serial Monitor displays an input field at the top, consisting of a single row and a Send button, and an output window below it, where data from the Arduino is displayed. When the Autoscroll box is checked, the most recent output is displayed, and once the screen is full, older data rolls off the screen as newer output is received. If you uncheck Autoscroll, you can manually examine the data using a vertical scroll bar.



Displaying Data from the Arduino in the Serial Monitor

- **Starting the Serial Monitor**: Before we can use the Serial Monitor, we need to activate it by adding this function to our sketch in void setup().
- The value 9600 is the speed at which the data will travel between the computer and the Arduino, also known as baud. This value must match the speed setting at the bottom right of the Serial Monitor.

```
Serial.begin(9600);
```

- **Sending Text to the Serial Monitor**: To send text to the Serial Monitor to be displayed in the output window, you can use Serial.print. This sends the text between the quotation marks to the Serial Monitor's output window.

```
Serial.print("Arduino for Everyone!");
```

Displaying Data from the Arduino in the Serial Monitor

- You can also use Serial.println to display text and then force any following text to start on the next line:

```
Serial.println("Arduino for Everyone!");
```

- **Displaying the Contents of Variables**: You can also display the contents of variables on the Serial Monitor. For example, this would display the contents of the variable results.

```
Serial.println(results);
```

- If the variable is a float, the display will default to two decimal places. You can specify the number of decimal places used as a number between 0 and 6 by entering a second parameter after the variable name. For example, to display the float variable results to four decimal places.

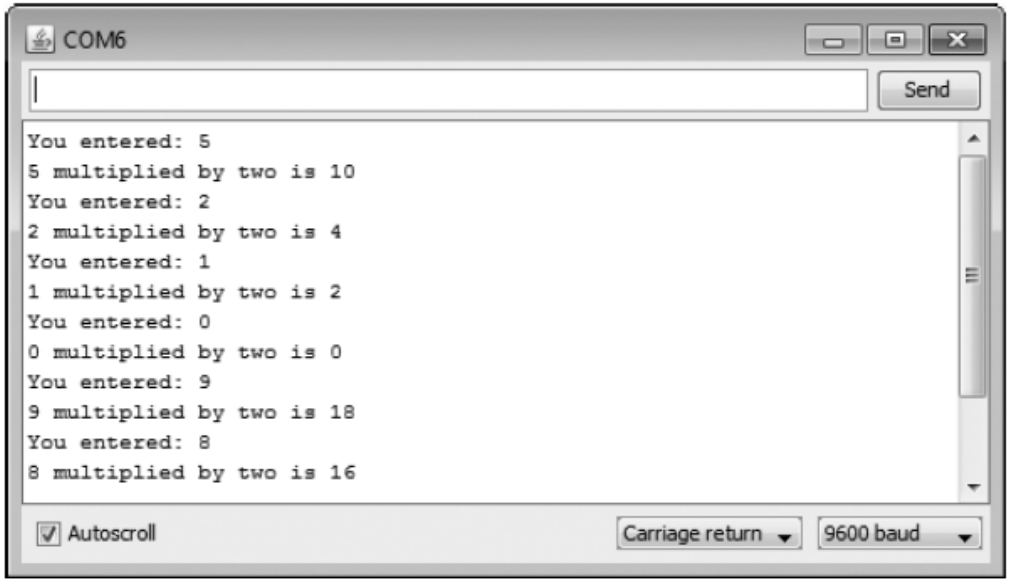
```
Serial.println(results);
```

Sending Data from the Serial Monitor to the Arduino

- To send data from the Serial Monitor to the Arduino, we need the Arduino to listen to the serial buffer.
- Example: Multiplying a Number by Two. To demonstrate the process of sending and receiving data via the Serial Monitor, let's dissect the following sketch. This sketch accepts a single digit from the user, multiplies it by 2, and then displays the result in the Serial Monitor's output window.
- The Serial.available() test in the first while statement at u returns 0 if nothing is entered yet into the Serial Monitor by the user. In other words, it tells the Arduino, "Do nothing until the user enters something." The next while statement at v detects the number in the serial buffer and converts the text code that represents the data entered into an actual integer number. Afterward, the Arduino displays the number from the serial buffer and the multiplication results.
- The Serial.flush() function at the start of the sketch clears the serial buffer just in case any unexpected data is in it, readying it to receive the next available data.

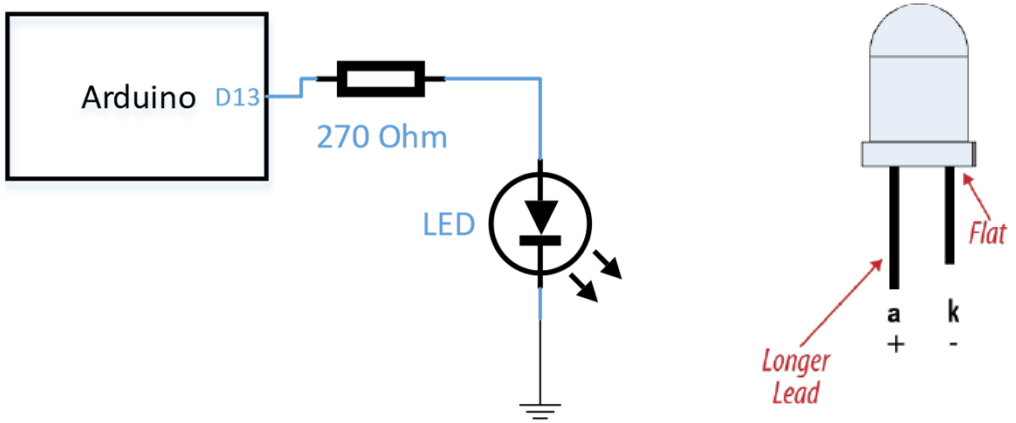
Sending Data from the Serial Monitor to the Arduino

```
int number;

void setup()
{
  Serial.begin(9600);
}
void loop()
{
  number = 0;     // zero the incoming number ready for a new read
  Serial.flush(); // clear any "junk" out of the serial buffer before waiting
❶ while (Serial.available() == 0)
  {
    // do nothing until something enters the serial buffer
  }
❷ while (Serial.available() > 0)
  {
    number = Serial.read() - '0';
// read the number in the serial buffer,
// remove the ASCII text offset for zero: '0'
  }
  // Show me the number!
  Serial.print("You entered: ");
  Serial.println(number);
  Serial.print(number);
  Serial.print(" multiplied by two is ");
  number = number * 2;
  Serial.println(number);
}
```

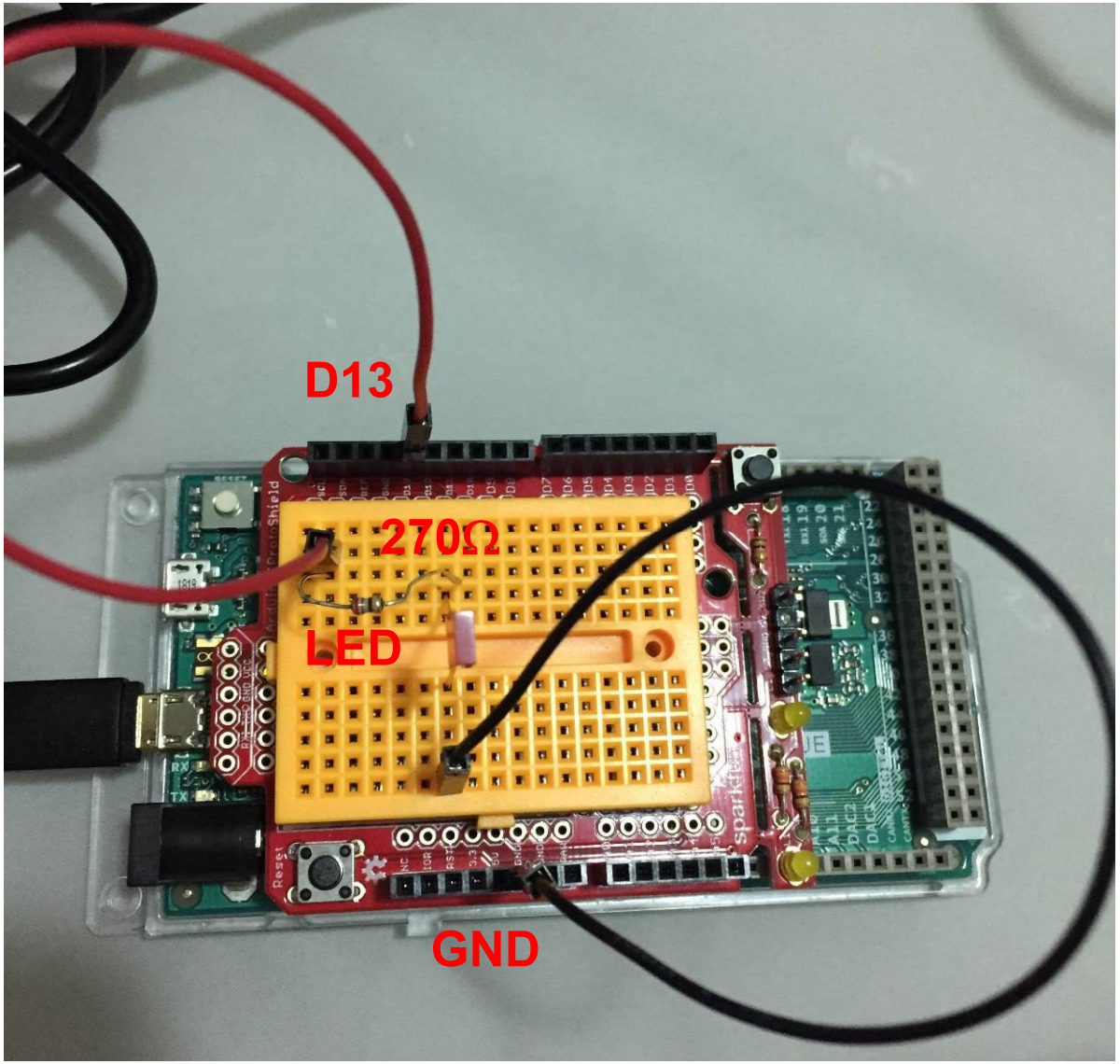


2. GPIO Output – external LED

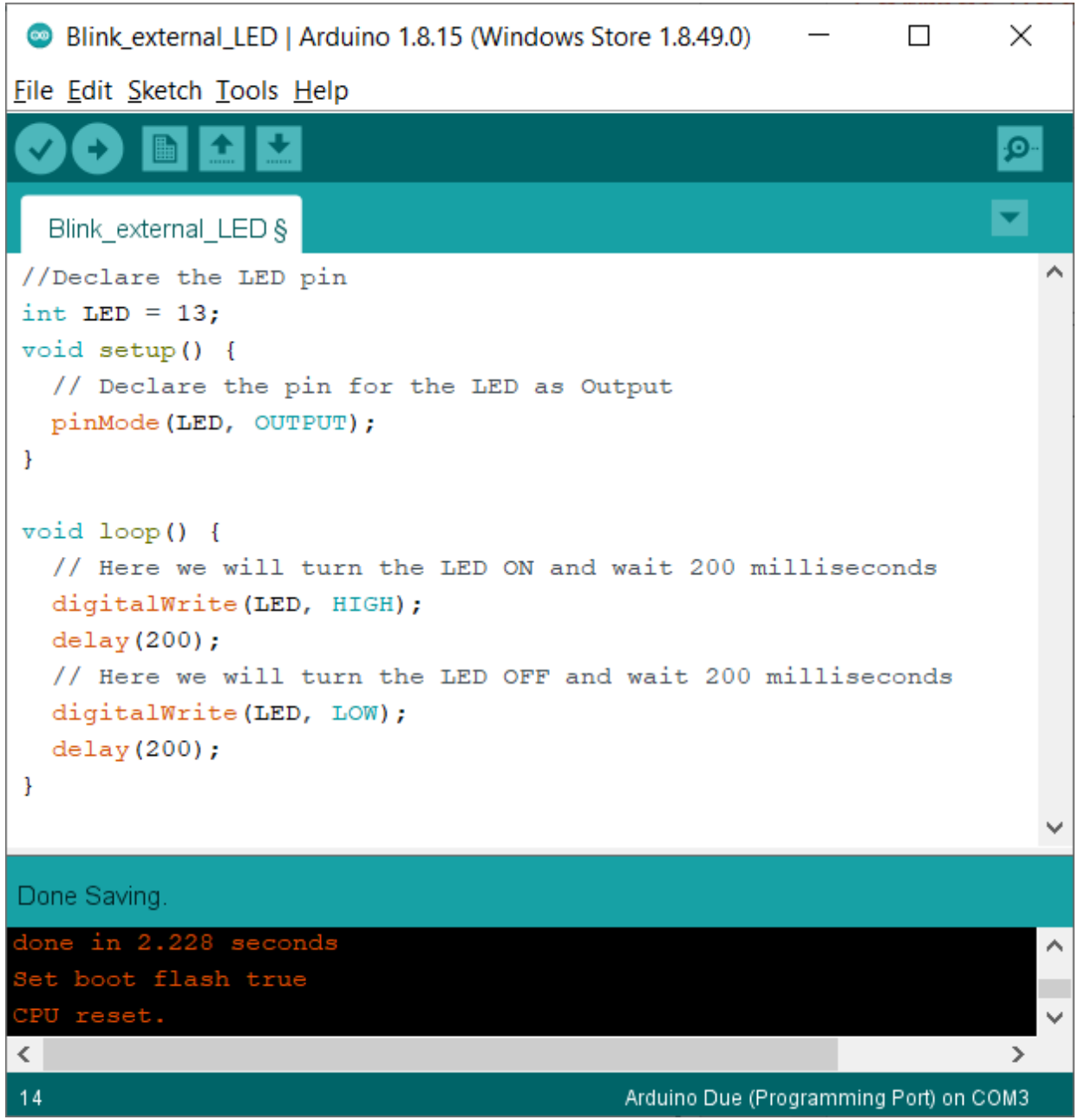
- we need the following component:
  1. An Arduino board connected to the computer via USB
  2. A breadboard and jumper wires
  3. A regular LED
  4. A resistor between 220–1,000 ohm
- Mount the resistor on the breadboard. Connect one end of the resistor to a digital pin on the Arduino board using a jumper wire.
- Mount the LED on the breadboard. Connect the anode (+) pin of the LED to the available pin on the resistor. We can determine the anode on the LED in two ways. Usually, the longer pin is the anode. Another way is to look for the flat edge on the outer casing of the LED. The pin next to the flat edge is the cathode (-).
- Connect the LED cathode (-) to the Arduino GND using jumper wires.



GPIO Output – external LED



GPIO Output – external LED



GPIO Output – Connecting an external LED

- How it works: when the second digital pin is set to HIGH, the Arduino provides 3.3 or 5 V of electricity, which travels through the resistor to the LED and GND.
- When enough voltage and current is present, the LED will light up. The resistor limits the amount of current passing through the LED.
- Without it, it is possible that the LED (or worse, the Arduino pin) will burn.
- Try to avoid using LEDs without resistors; this can easily destroy the LED or even your Arduino.

GPIO Output – external LED

- Code breakdown:
  - The code simply turns the LED on, waits, and then turns it off again.
  - We will use a blocking approach by using the delay() function.
  - Here we declare the LED pin on digital pin 13 for Arduino Due;
  - In the setup() function we set the LED pin as an output:

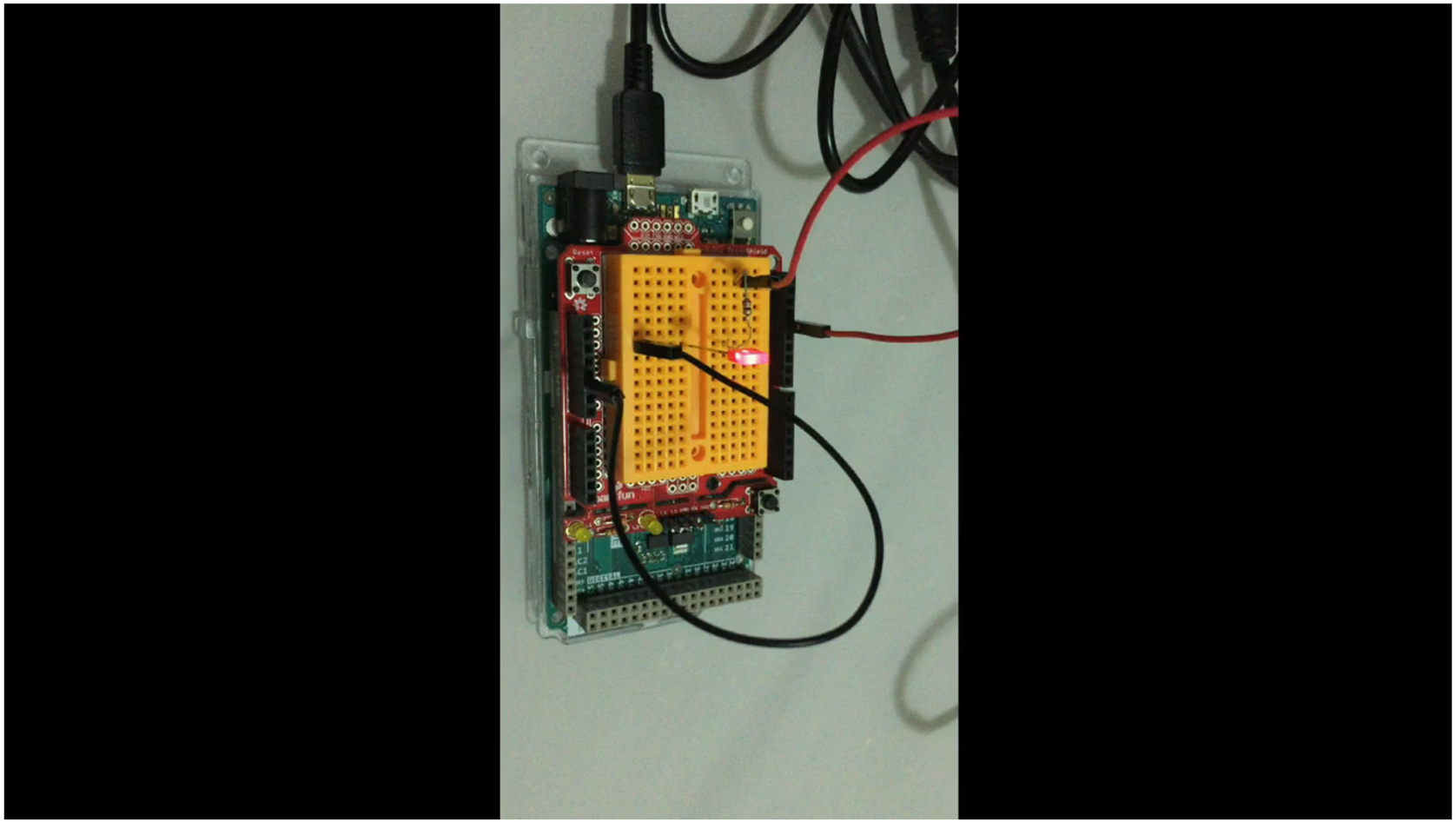
```
//Declare the LED pin
int LED = 13;
void setup() {
  // Declare the pin for the LED as Output
  pinMode(LED, OUTPUT);
}
```

GPIO Output – external LED

- In the loop() function, we continuously turn the LED on, wait 200 milliseconds, and then we turn it off.
- After turning it off we need to wait another 200 milliseconds, otherwise it will instantaneously turn on again and we will only see a permanently on LED.

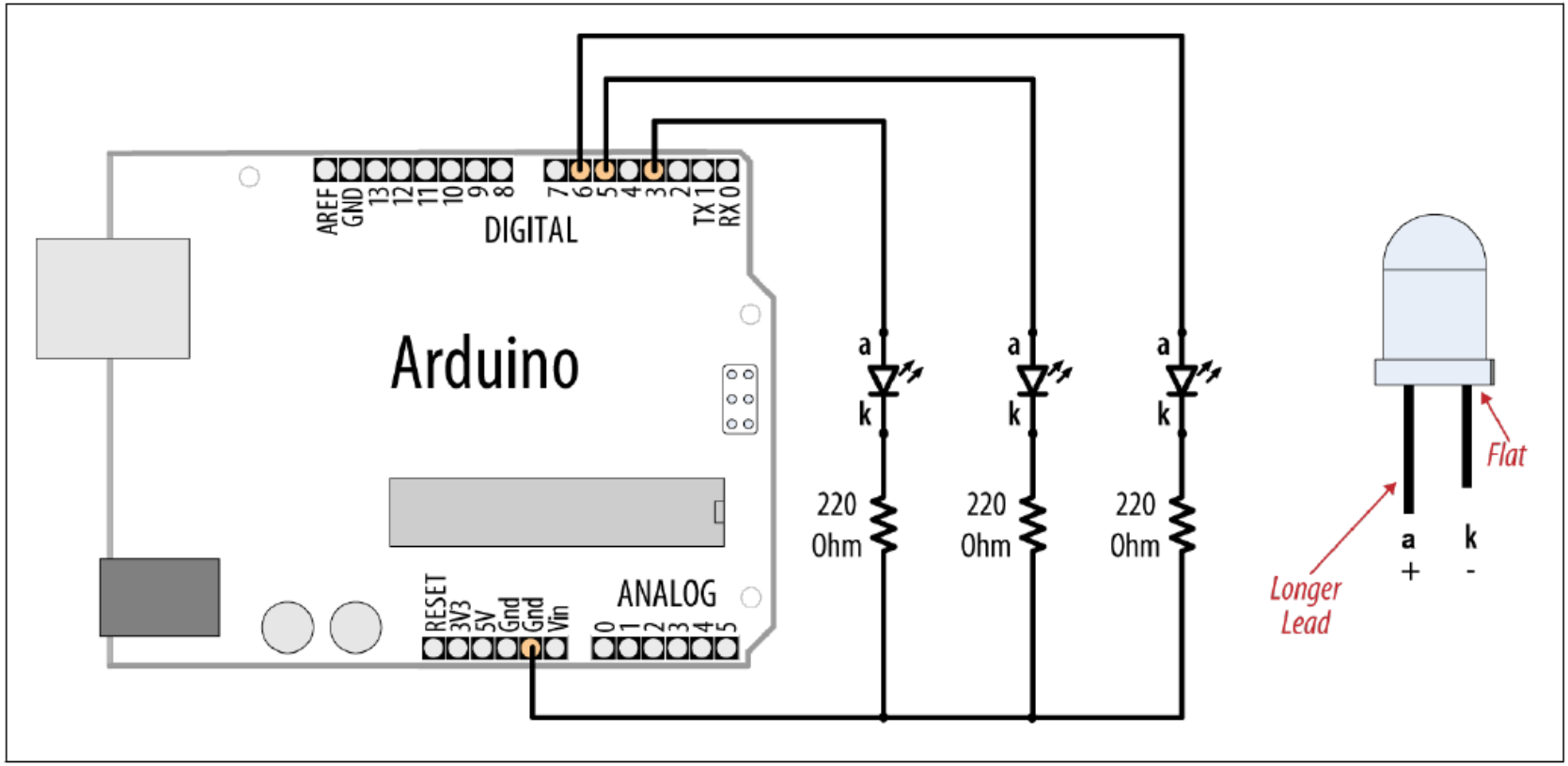
```
void loop() {
  // Here we will turn the LED ON and wait 200 milliseconds
  digitalWrite(LED, HIGH);
  delay(200);
  // Here we will turn the LED OFF and wait 200 milliseconds
  digitalWrite(LED, LOW);
  delay(200);
}
```

GPIO Output – external LED



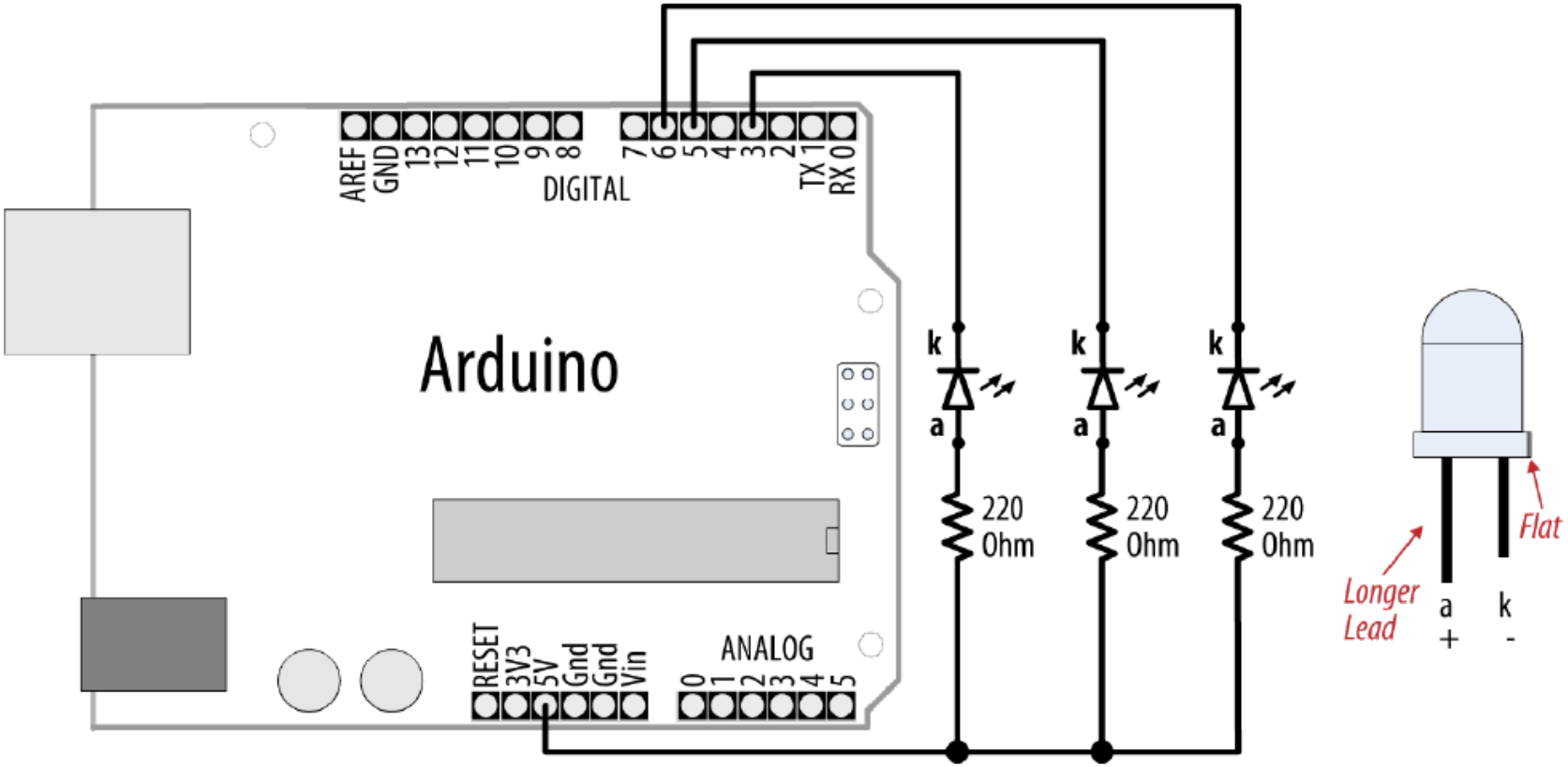
GPIO Output – Multiple LEDs

- Connecting external LEDs with the Anode connected to pins of embedded system

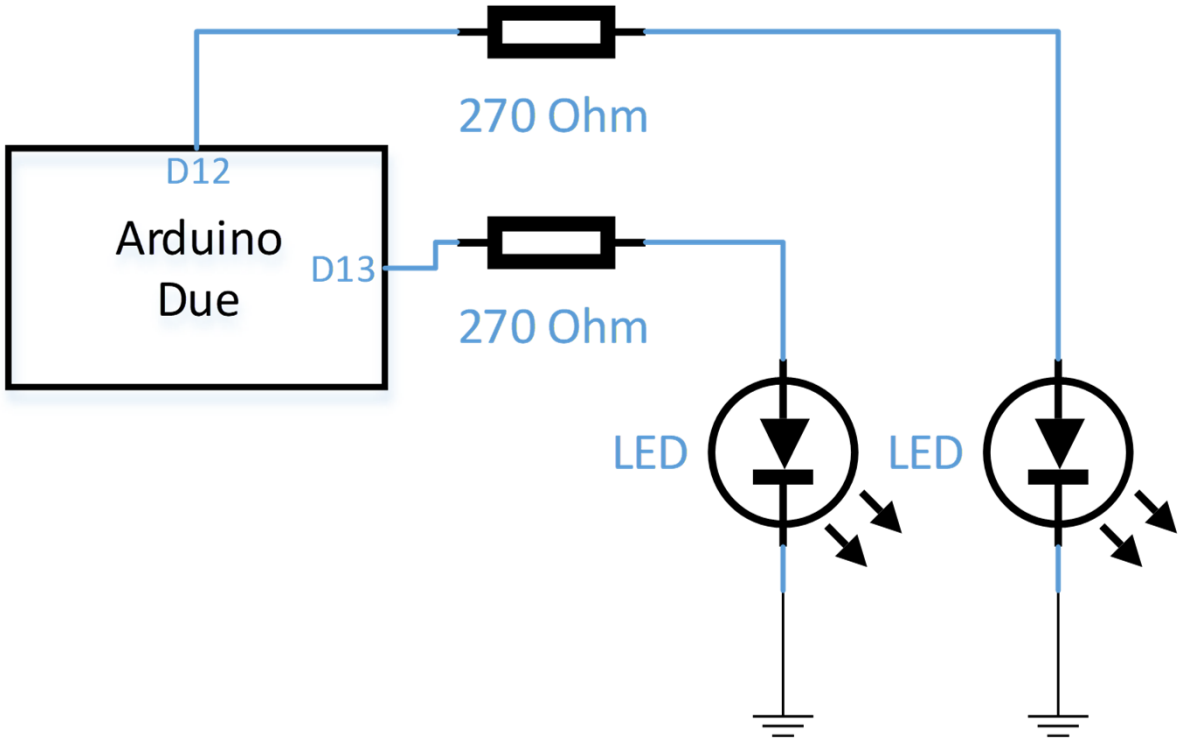


GPIO Output – Multiple LEDs

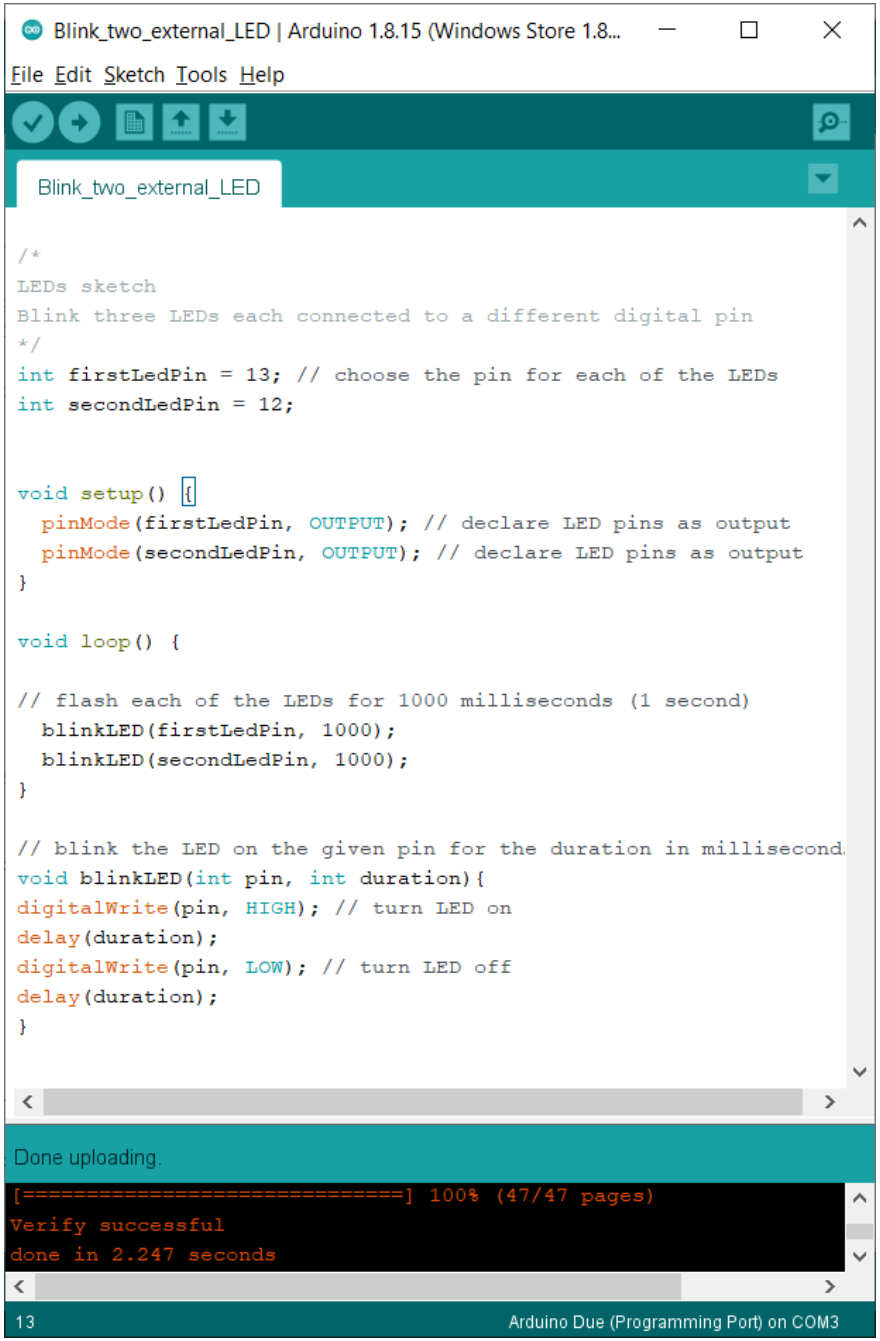
- Connecting external LEDs with the Chatode connected to pins of embedded system



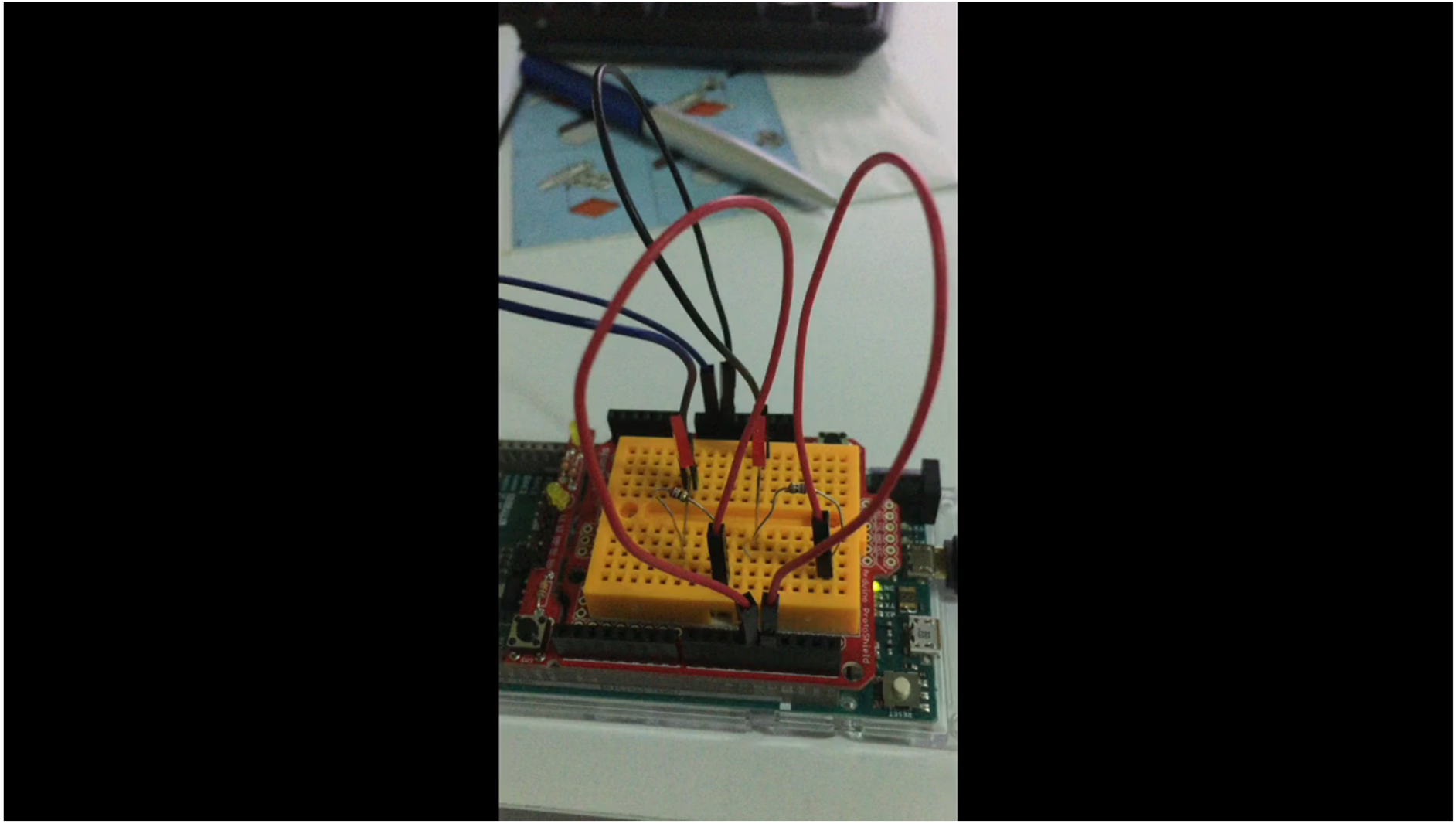
GPIO Output – Multiple LEDs



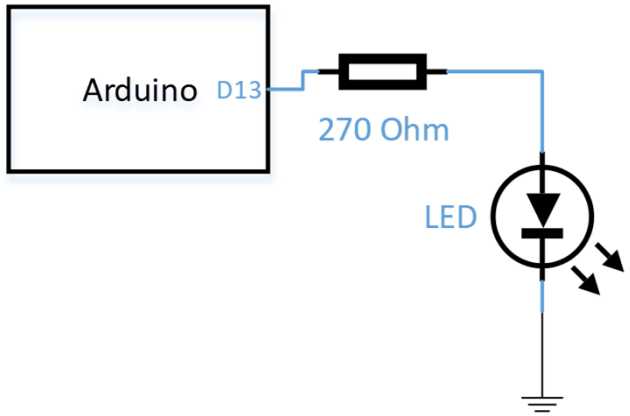
GPIO Output – Multiple LEDs



GPIO Output – Multiple LEDs



3. GPIO Output – LED with PWM



LED_PWM | Arduino 1.8.15 (Windows Store 1.8....

File Edit Sketch Tools Help

LED_PWM

```
// Declare the LED pin with PWM
int LED = 13;

void setup() {
// Declare the pin for the LED as Output
  pinMode(LED, OUTPUT);
}

void loop() {
// Here we will fade the LED from 0 to maximum, 255
  for (int i = 0; i < 256; i++){
    analogWrite(LED, i);
    delay(5);
  }
// Fade the LED from maximum to 0
  for (int i = 255; i >= 0; i--){
    analogWrite(LED, i);
    delay(5);
  }
}
```

Done uploading.

```
done in 5.053 seconds
Set boot flash true
CPU reset.
```

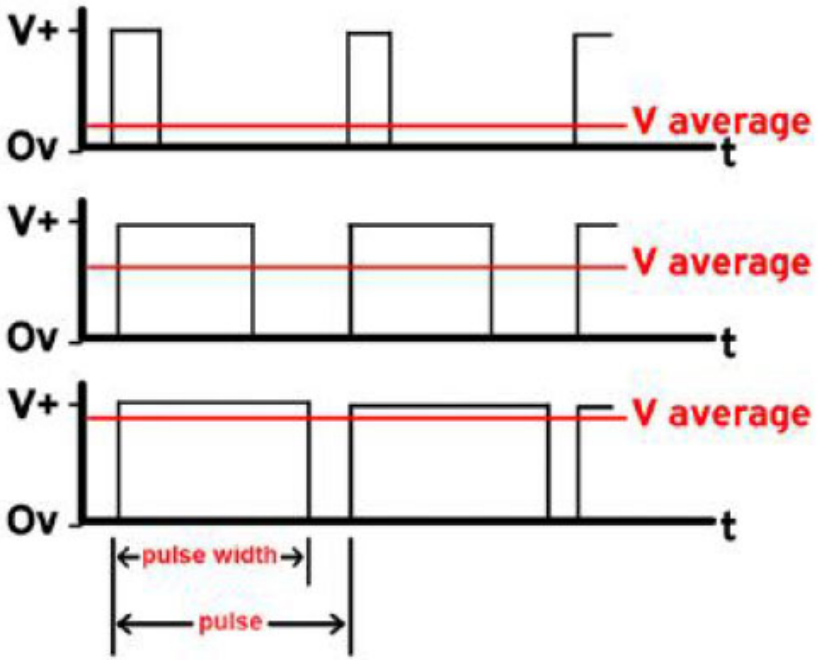
20 Arduino Due (Programming Port) on COM3

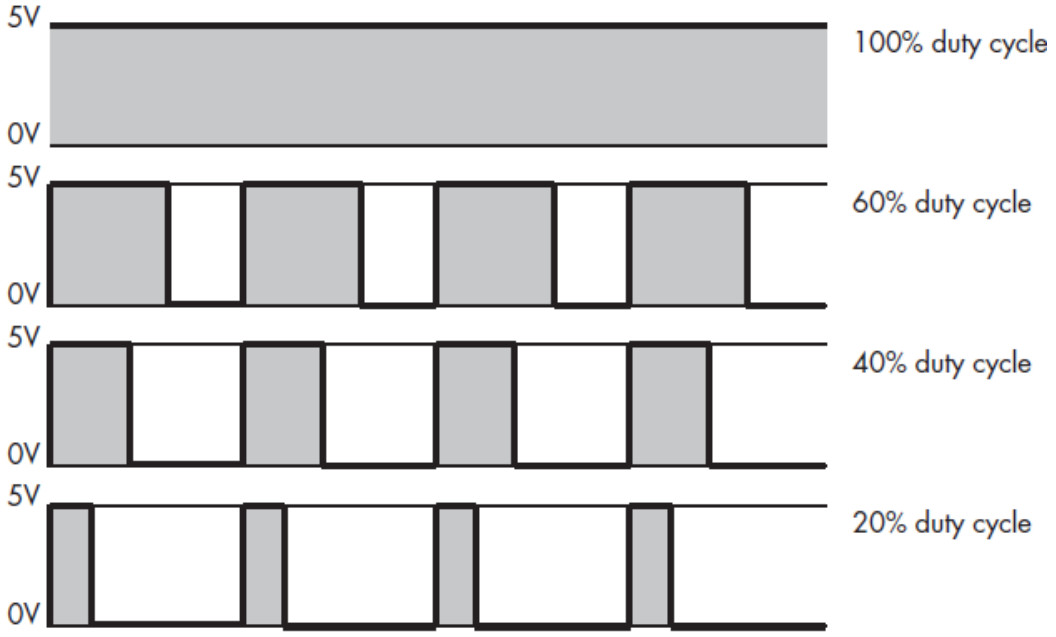
GPIO Output – LED with PWM

- Rather than just turning LEDs on and off rapidly using digitalWrite(), we can define the level of brightness of an LED by adjusting the amount of time between each LED's on and off states using pulse-width modulation (PWM).
- The brightness we perceive is determined by the amount of time the digital output pin is on versus the amount of time it is off—that is, every time the LED is lit or unlit. Because our eyes can't see flickers faster than 50 cycles per second, the LED appears to have a constant brightness.
- The greater the duty cycle (the longer the pin is on compared to off in each cycle), the greater the perceived brightness of the LED connected to the digital output pin.
- This all works with Pulse Width Modulation (PWM), which works by switching between LOW and HIGH very fast.
- If we turn a digital pin on and off a thousand times per second, we will obtain, on average, a voltage that is half of the HIGH voltage.
- If the ratio between HIGH and LOW is 2:3, the obtained voltage will be two-thirds of the HIGH voltage and so on.

GPIO Output – LED with PWM

- The following diagrams better explains how PWM works and various PWM duty cycles:



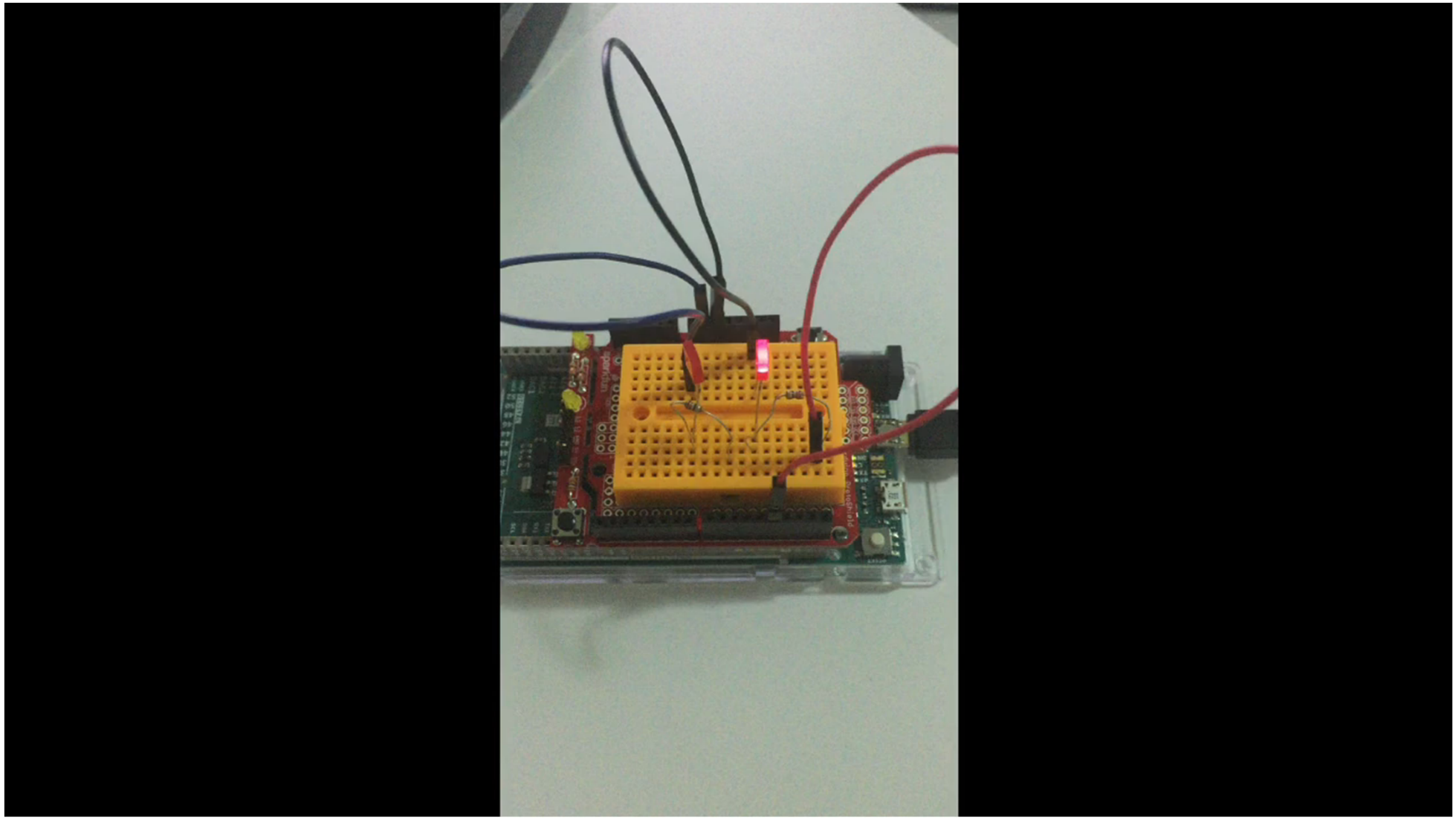


GPIO Output – LED with PWM

- In the loop() function, we use the important PWM function analogWrite().
- This function provides an analog signal on the digital PWM pin.
- The values for the voltage can be between 0–255, 0 for 0 volts and 255 for 5 V or 3.3 V, depending on the Arduino board used.
- Here, we fade in the LED slowly using a for function and then we fade it out:

```
void loop(){
  // Here we will fade the LED from 0 to maximum, 255
  for (int i = 0; i < 256; i++){
    analogWrite(LED, i);
    delay(5);
  }
  // Fade the LED from maximum to 0
  for (int i = 255; i >= 0; i--){
    analogWrite(LED, i);
    delay(5);
  }
}
```

GPIO Output – LED with PWM

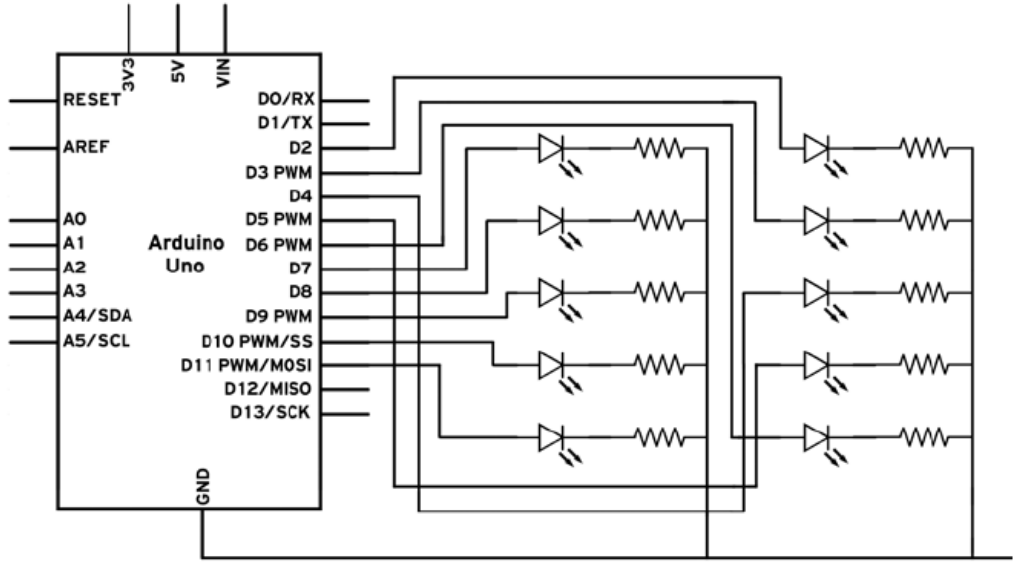


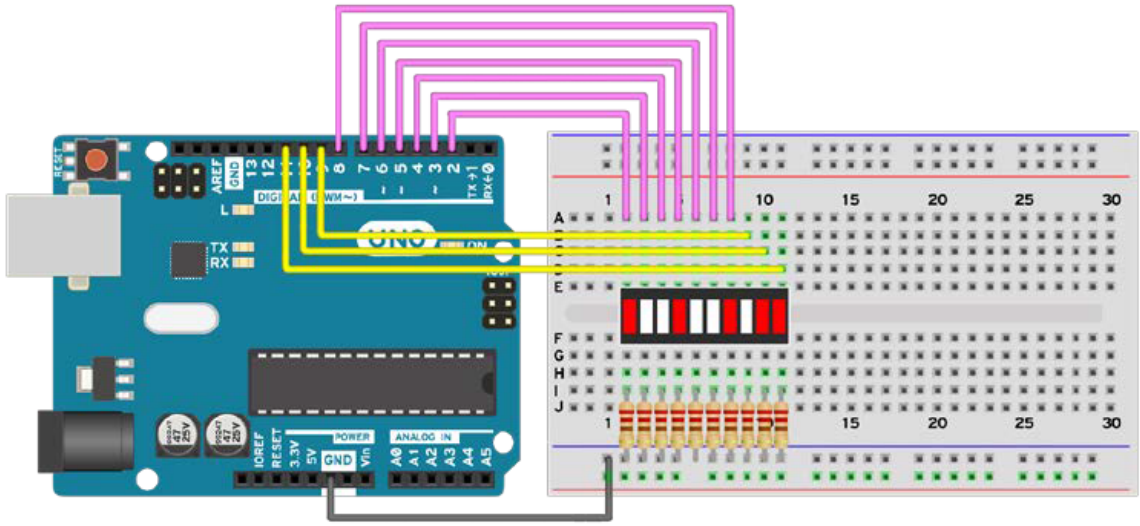
4. GPIO Output – LED bar graph

- An LED bar graph is just a bunch of LEDs put together in a fancy case, but there are many uses for it.
- We can display the date from a sensor, show a critical condition, or make a funny light show with it, etc.
- Following are the steps to connect a 10-segment bar graph to the Arduino:
  - Mount the LED bar graph onto the breadboard.
  - If the bar graph is a common anode (+) configuration, connect the common anode (+) pin to the 5V/3.3V port on the Arduino. If the bar graph is a common cathode (-), connect the pin to the GND port on the Arduino.
  - Connect each individual segment pin to one individual Arduino digital pin, using a resistor. To make things simple, connect all the segment pins to successive digital pins on the Arduino.

GPIO Output – LED bar graph

- Example connection of a common anode (+) 10-segment LED bar graph:





GPIO Output – LED bar graph

- The following code will make the LED bar graph full and then empty (designed for for a common anode (+) configuration)

```
// Declare the first and last Pin of the LED Bar
int pin1 = 2;
int pin10 = 11;

void setup() {
  // Declare the pins as Outputs
  for (int i = pin1; i <= pin10; i++){
    pinMode(i, OUTPUT);
  }
}

// A simple function to set the value of the LED Bar
void setBarValue(int value){
  // First we turn everything off
  for (int i = pin1; i <= pin10; i++){
    digitalWrite(i, HIGH);
  }

  // Write the value we want
  for (int i = pin1; i <= pin1 + value; i++){
    digitalWrite(i, LOW);
  }

  // In case we have value 0
  if (value == 0){
  digitalWrite(pin1, HIGH);
  }
}

void loop(){
  // Play with a few displays

  // Ping-Pong
  for (int i = 0; i <= 10; i++){
    setBarValue(i);
    delay(100);
  }
 for (int i = 10; i >= 0; i--){
    setBarValue(i);
    delay(100);
  }
}
```

GPIO Output – LED bar graph

- Codes breakdown: This code loads and unloads the LED bar graph just like a progress bar. Here, we declare the first and the last pins used in the LED bar. There is no point in declaring all of them as we know they are consecutive in this implementation:

```
int pin1 = 2;
int pin10 = 11;
```

- In the setup() function, we set each LED pin as an output. This simple trick, used here, helps to set all the pins between pin1 and pin10 as outputs:

```
void setup() {
  // Declare the pins as Outputs
  for (int i = pin1; i <= pin10; i++){
    pinMode(i, OUTPUT);
  }
}
```

GPIO Output – LED bar graph

- In the custom setBarValue() function, we make the bar show a certain progress level. As the maximum is 10 and the minimum is 0, if we write 5, half the LEDs on the bar will be on while the other half are off:

```
// A simple function to set the value of the LED Bar
void setBarValue(int value){
  // First we turn everything off
  for (int i = pin1; i <= pin10; i++){
    digitalWrite(i, HIGH);
  }

  // Write the value we want
  for (int i = pin1; i <= pin1 + value; i++){
    digitalWrite(i, LOW);
  }

  // In case we have value 0
  if (value == 0) digitalWrite(pin1, HIGH);
}
```

GPIO Output – LED bar graph

- Finally, in the loop() function, we use our custom function to load the bar and then unload it. In the following code, we use a for loop to increase the bar value to the maximum and then we decrease it back to 0:

```
void loop(){
  for (int i = 0; i <= 10; i++){
    setBarValue(i);
    delay(100);
  }
 for (int i = 10; i >= 0; i--){
    setBarValue(i);
    delay(100);
  }
}
```

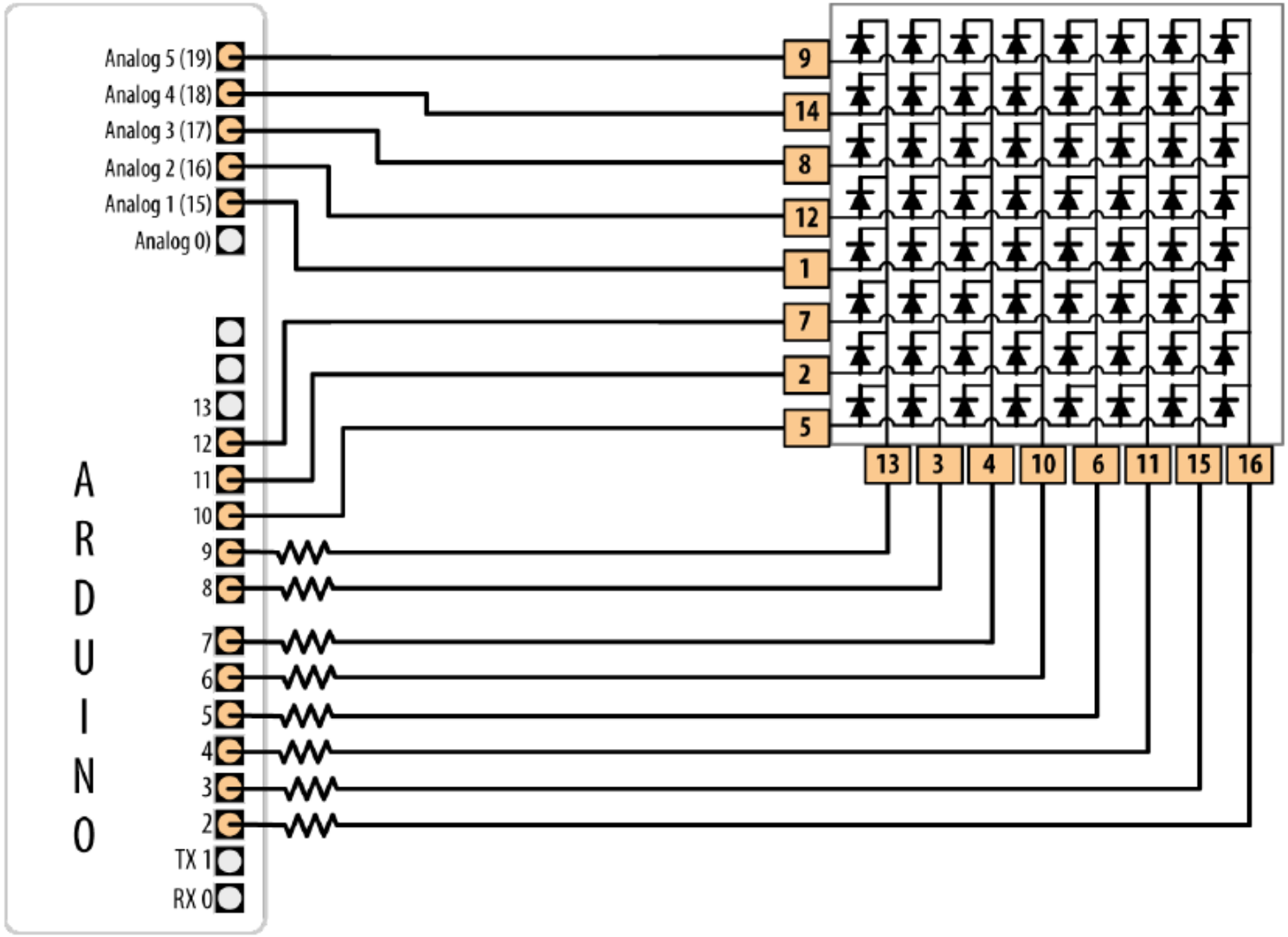
- Variations of LED bar graph: Common anode (+) and common cathode (-)
- Each LED bar is either a common anode (+) or a common cathode (-).
- If it's a common anode (+), we connect the anode to 5V/3.3V, each other pin to a resistor, and the resistors to individual digital pins on the Arduino.
- For the common cathode (-), connect the cathode to the GND and each pin, using a resistor, to individual Arduino digital pins.

5. GPIO Output – LED Matrix

- If there is a matrix of LEDs and want to minimize the number of Arduino pins needed to turn LEDs on and off, one of the solutions is by *controlling an LED Matrix using multiplexing*.
- The following example uses an LED matrix of 64 LEDs, with anodes connected in rows and cathodes in columns.
- LED matrix displays do not have a standard pinout, so you must check the data sheet for your display. Wire the rows of anodes and columns of cathodes as shown in the following figure but use the LED pin numbers shown in your data sheet.
- The resistor's value must be chosen to ensure that the maximum current through a pin does not exceed 40 mA. Because the current for up to eight LEDs can flow through each column pin, the maximum current for each LED must be one-eighth of 40 mA, or 5 mA.
- Each column of the matrix is connected through the series resistor to a digital pin. When the column pin goes low and a row pin goes high, the corresponding LED will light. For all LEDs where the column pin is high or its row pin is low, no current will flow through the LED and it will not light.

GPIO Output – LED Matrix

- Example of the schematic/diagram circuit for LED Matrix:



GPIO Output – LED Matrix

- Example of coding for LED Matrix:

```
/*
  matrixMpx sketch

 Sequence LEDs starting from first column and row until all LEDS are lit
 Multiplexing is used to control 64 LEDs with 16 pins
 */

const int columnPins[] = { 2, 3, 4, 5, 6, 7, 8, 9};
const int rowPins[]   = { 10,11,12,15,16,17,18,19};

int pixel       = 0;        // 0 to 63 LEDs in the matrix
int columnLevel = 0;        // pixel value converted into LED column
int rowLevel    = 0;        // pixel value converted into LED row

void setup() {
  for (int i = 0; i < 8; i++)
  {
    pinMode(columnPins[i], OUTPUT);  // make all the LED pins outputs
    pinMode(rowPins[i], OUTPUT);
  }
}
```

GPIO Output – LED Matrix

- Example of coding for LED Matrix:

```
void loop() {
  pixel = pixel + 1;
  if(pixel > 63)
     pixel = 0;

  columnLevel = pixel / 8;                    // map to the number of columns
  rowLevel = pixel % 8;                       // get the fractional value
  for (int column = 0; column < 8; column++)
  {
    digitalWrite(columnPins[column], LOW);    // connect this column to Ground
    for(int row = 0; row < 8; row++)
    {
      if (columnLevel > column)
      {
        digitalWrite(rowPins[row], HIGH);  // connect all LEDs in row to +5 volts
      }
      else if (columnLevel == column && rowLevel >= row)
      {
          digitalWrite(rowPins[row], HIGH);
      }
      else
      {
        digitalWrite(columnPins[column], LOW); // turn off all LEDs in this row
      }
      delayMicroseconds(300);     // delay gives frame time of 20ms for 64 LEDs
      digitalWrite(rowPins[row], LOW);         // turn off LED
    }

    // disconnect this column from Ground
    digitalWrite(columnPins[column], HIGH);
  }
}
```

GPIO Output – LED Matrix

- The for loop scans through each row and column and turns on sequential LEDs until all LEDs are lit.
- The loop starts with the first column and row and increments the row counter until all LEDs in that row are lit; it then moves to the next column, and so on, lighting another LED with each pass through the loop until all the LEDs are lit.
- You can control the number of lit LEDs in proportion to the value from a sensor by making the following changes to the sketch/Arduino program. Comment out or remove these three lines from the beginning of the loop:

```
pixel = pixel + 1;
if(pixel > 63)
   pixel = 0;
```

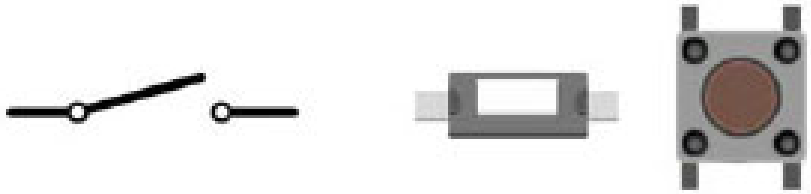
- Replace them with the following lines that read the value of a sensor on pin 0 and map this to a number of pixels ranging from 0 to 63:

```
int sensorValue = analogRead(0);              // read the analog in value
pixel =   map(sensorValue, 0, 1023, 0, 63);   // map sensor value to pixel (LED)
```

- The number of LEDs lit will be proportional to the value of the sensor.

GPIO Input – Button

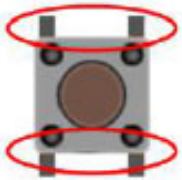
- Buttons are the basis of human interaction with the Arduino. We press a button, and something happens. They are simple components, as they only have two states: opened or closed. When a button is closed, current can pass though it. When it's opened, no current can pass. Some buttons are closed when we push them, some when they are released.
- Momentary buttons are active as long as they are pressed, while maintained buttons keep the state we let them in. Keyboards have momentary buttons while the typical light switch is a maintained button.
- Momentary buttons can either be opened or closed. This reflects the connection state when not pressed. A closed momentary switch will conduct current while not pressed and interrupt the current when pressed. An opened button will do the opposite. Lastly, there are poles and throws. The following figure explains the main two types:



6. GPIO Input – Button

- Single Pole Single Throw (SPST) switches have a closed state in which they conduct current, and an opened state in which they do not conduct current. Most momentary buttons are SPST.

- Single Pole Double Throw (SPDT) switches route the current from the common pin to one of the two outputs. They are typically maintained; one example of this is the common light switch.
- The common button we'll be using in this chapter is a push button. It's a small momentary opened switch. It typically comes in a 4-pin case.
- The pins inside the two red ellipses are shorted together. When we press the button, all four pins are connected.

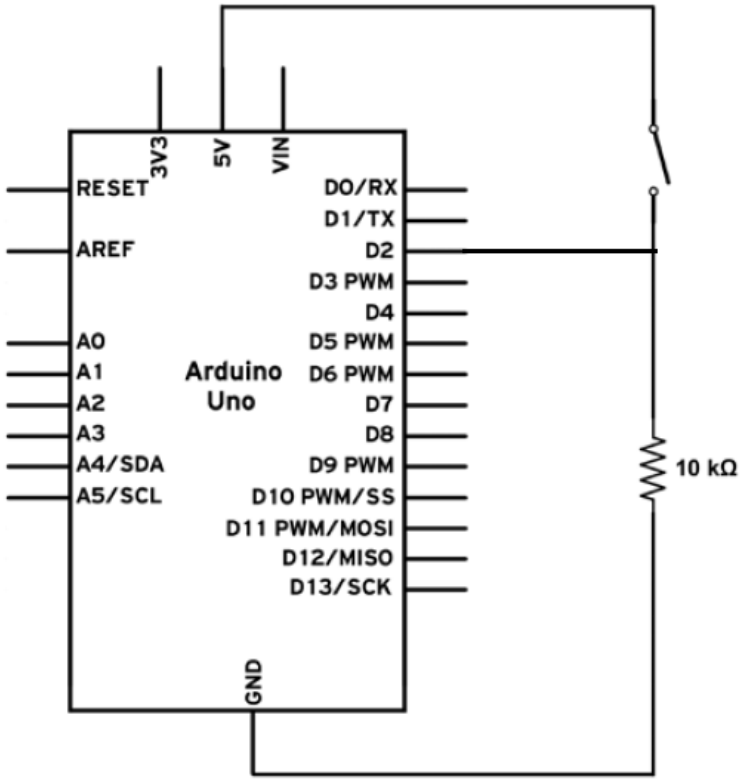


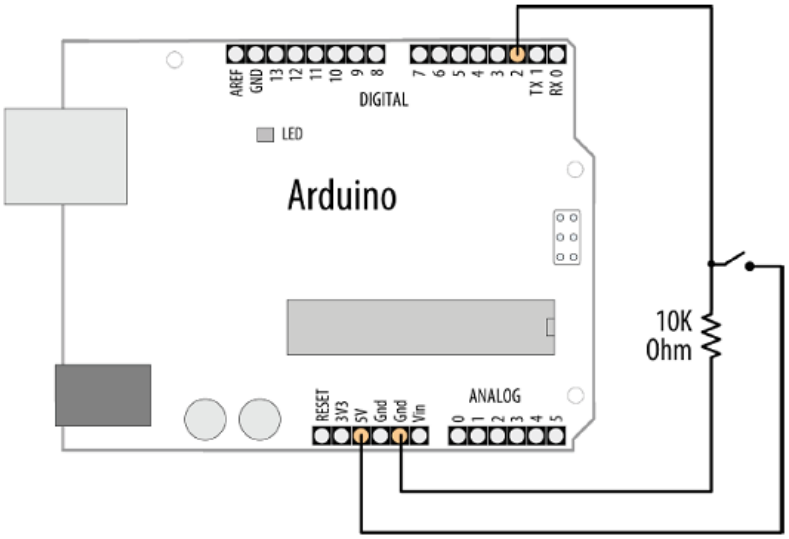
GPIO Input – Button

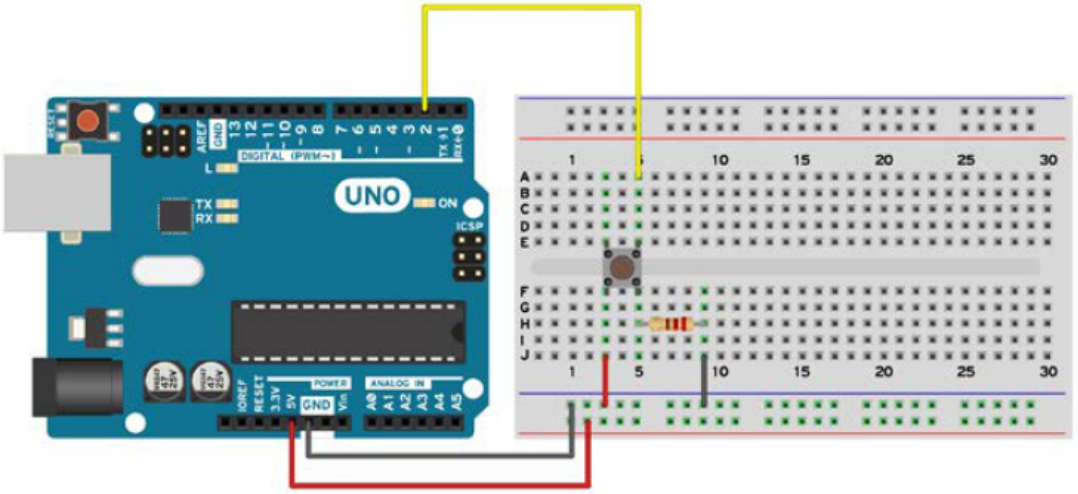
- The following is example of connecting a button:
  - Connect the Arduino GND and 5V/3.3V to separate long strips on the breadboard.
  - Mount the push button on the breadboard and connect one terminal to the 5V/3.3V long strip and the other to a digital pin on the Arduino—in this example, pin 2.
  - Mount the resistor between the chosen digital pin and the GND strip. This iscalled a pull-down setup. More on this later.

GPIO Input – Button

- Schematic for connecting button:







GPIO Input – Button

- The following code will read if the button has been pressed and will control the built-in LED.
- If the button is connected to a different pin, simply change the buttonPin value to the value of the pin that has been used.
- The purpose of the button is to drive the digital pin to which it's connected to either HIGH or LOW. In theory, this should be very simple: just connect one end of the button to the pin and the other to 5V. When not pressed, the voltage will be LOW; otherwise it will be 5V, HIGH.
- However, there is a problem. When the button is not pressed, the input will not be LOW but instead a different state called floating. In this state, the pin can be either LOW or HIGH depending on interference with other components, pins, and even atmospheric conditions.
- That's where the resistor comes in. It is called a pull-down resistor as it pulls the voltage down to GND when the button is not pressed. This is a very safe method when the resistor value is high enough. Any value over 1K will work just fine, but 10K ohm is recommended.

GPIO Input – Button

- Codes:

```
// Declare the pins for the Button and the LED
int buttonPin = 2;
int LED = 13;

void setup() {
  // Define pin #2 as input
  pinMode(buttonPin, INPUT);
  // Define pin #13 as output, for the LED
  pinMode(LED, OUTPUT);
}

void loop() {
  // Read the value of the input. It can either be 1 or 0.
  int buttonValue = digitalRead(buttonPin);

  if (buttonValue == HIGH) {
    // If button pushed, turn LED on
    digitalWrite(LED,HIGH);
  } else {
    // Otherwise, turn the LED off
    digitalWrite(LED, LOW);
  }
}
```

GPIO Input – Button

- The codes breakdown: the code takes the value from the button. If the button is pressed, it will start the built-in LED. Otherwise, it will turn it off.
- Here, we declare the pin to which the button is connected as pin 2, and the built-in LED on pin 13:

```
int buttonPin = 2;
int LED = 13;
```

- In the setup() function, we set the button pin as a digital input and the LED pin as an output:

```
void setup() {
  pinMode(buttonPin, INPUT);
  pinMode(LED, OUTPUT);
}
```

- The important part comes in the loop() function. The first step is to declare a variable that will equal the value of the button state. This is obtained using the digitalRead() function:

```
int buttonValue = digitalRead(buttonPin);
```

GPIO Input – Button

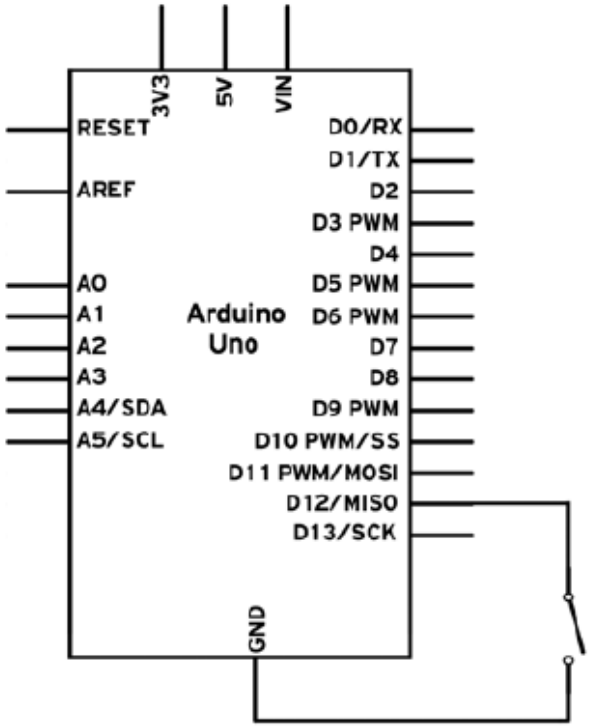
- Lastly, depending on the button state, we initiate another action. In this case, we just light up the LED or turn it off:

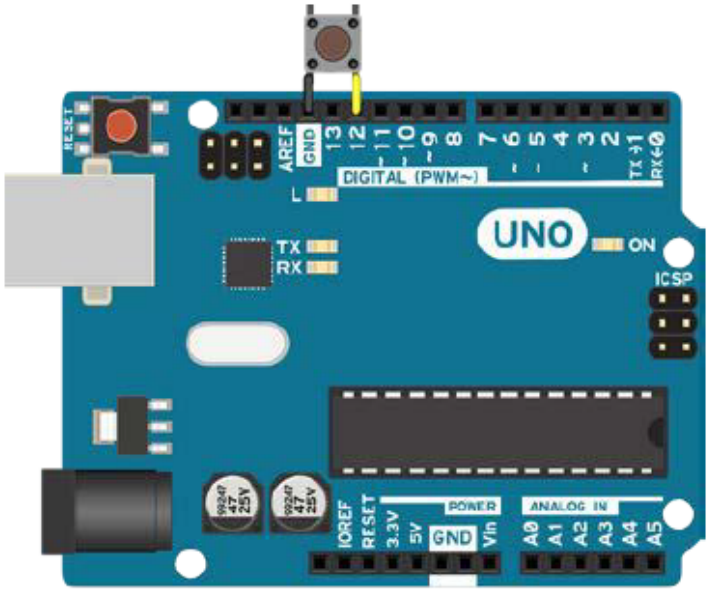
```
if (buttonValue == HIGH){
        digitalWrite(LED,HIGH);
  } else {
    // Otherwise, turn the LED off
    digitalWrite(LED, LOW);
  }
```

- In a pull-up configuration, the resistor will pull up the voltage to 5V when the button is not pressed. To implement it, connect one terminal of the button to the digital pin and the other one to GND. Now, connect the resistor between the digital pin and 5V. This configuration will return inverted values. When pressed, the button will give LOW, not HIGH, as it will draw the pin down to GND, 0 V. However, it brings no advantages over the pull-down configuration.

7. GPIO Input – Button with no resistor

- The resistor is mandatory for proper operation of a button, and everybody will insist on using it. However, there is a little secret embedded in each Arduino pin. Each pin already has a pull-up resistor that we can enable with just one small change in our code.
- The connection by Connect the Arduino GND to a terminal on the button and connect the chosen digital pin to the other terminal.





GPIO Input – Button with no resistor

- Codes: If the button is connected to a different pin, change the buttonPin value to the value of the pin that has been used.
- When we press the button, the value of the Arduino pin should be either LOW or HIGH. In this configuration, when we press the button, the pin is connected directly to GND, resulting in LOW.
- However, when it is not pressed, the pin will have no value; it will be in a floating state. To avoid this, an internal pull-up resistor is connected between each pin and 5V. When we activate the resistor, it will keep the pin at HIGH until we press the button, thus connecting the pin to GND.

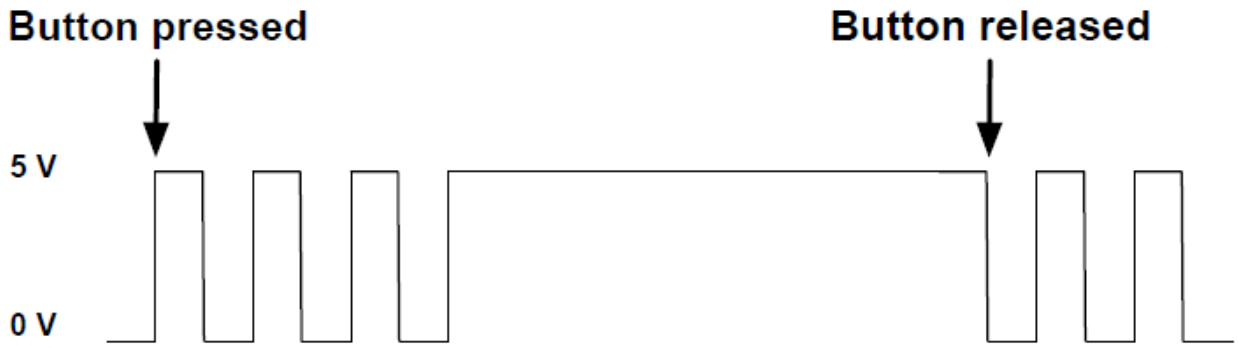
GPIO Input – Button with no resistor

- Codes:

```
// Declare the pins for the Button and the LED
int buttonPin = 12;
int LED = 13;

void setup() {
  // Define pin #12 as input and activate the internal pull-up
resistor
  pinMode(buttonPin, INPUT_PULLUP);
  // Define pin #13 as output, for the LED
  pinMode(LED, OUTPUT);
}

void loop(){
  // Read the value of the input. It can either be 1 or 0
  int buttonValue = digitalRead(buttonPin);

  if (buttonValue == LOW){
    // If button pushed, turn LED on
    digitalWrite(LED,HIGH);
  } else {
    // Otherwise, turn the LED off
    digitalWrite(LED, LOW);
  }
}
```

8. GPIO Input – Button with Debouncing

- You will still find some cases when the button doesn't behave fully as expected. Problems mainly occur in the moment you release the button.
- These problems occur because the mechanical buttons bounce for a few milliseconds when you press them. In the following figure, you can see a typical signal produced by a mechanical button. Right after you have pressed the button, it doesn't emit a clear signal.
- To overcome this effect, you have to debounce the button. It's usually sufficient to wait a short period of time until the button's signal stabilizes. Debouncing ensures that the input pin reacts only once to a push of the button:



8. GPIO Input – Button with Debouncing

- This final version of our LED switch differs from the previous one in only a single line: to debounce the button, we wait for 50 milliseconds in line 21 before we enter the main loop again.

BinaryDice/DebounceButton/DebounceButton.ino

```
Line 1  const unsigned int BUTTON_PIN = 7;
    -   const unsigned int LED_PIN    = 13;
    -   void setup() {
    -     pinMode(LED_PIN, OUTPUT);
    5     pinMode(BUTTON_PIN, INPUT);
    -   }
    -
    -   int old_button_state = LOW;
    -   int led_state = LOW;
   10
    -   void loop() {
    -     const int CURRENT_BUTTON_STATE = digitalRead(BUTTON_PIN)
    -     if (CURRENT_BUTTON_STATE != old_button_state &&         ;
    -         CURRENT_BUTTON_STATE == HIGH)
   15     {
    -       if (led_state == LOW)
    -         led_state = HIGH;
    -        else
    -         led_state = LOW;
   20       digitalWrite(LED_PIN, led_state);
    -       delay(50);
    -     }
    -     old_button_state = CURRENT_BUTTON_STATE;
    -   }
```

Seminar 2– ADC

Topics:

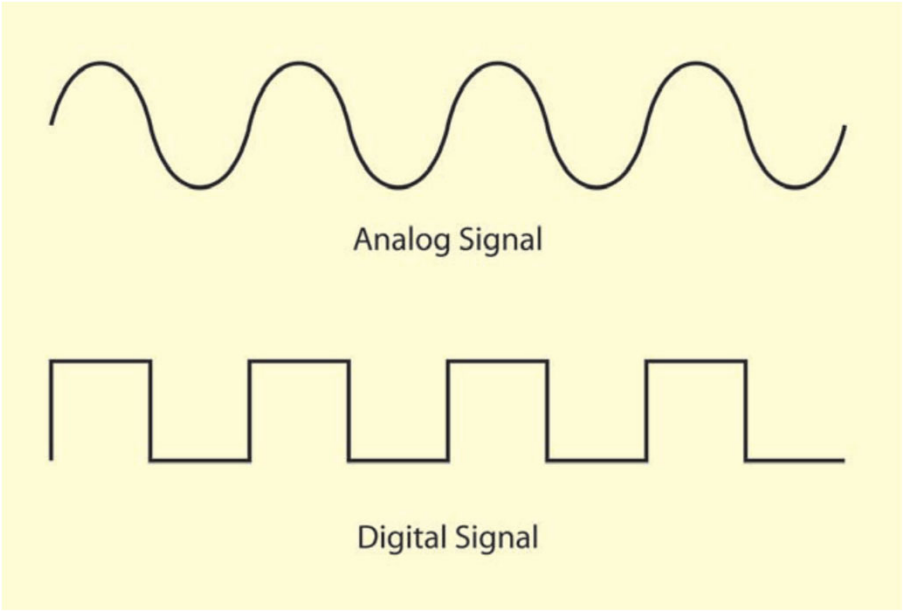
1. Digital vs Analog
2. ADC Theory
3. Arduino – ADC Specification
4. Potentiometer
5. Light Sensor
6. Microphone - Sound

1. Digital vs Analog

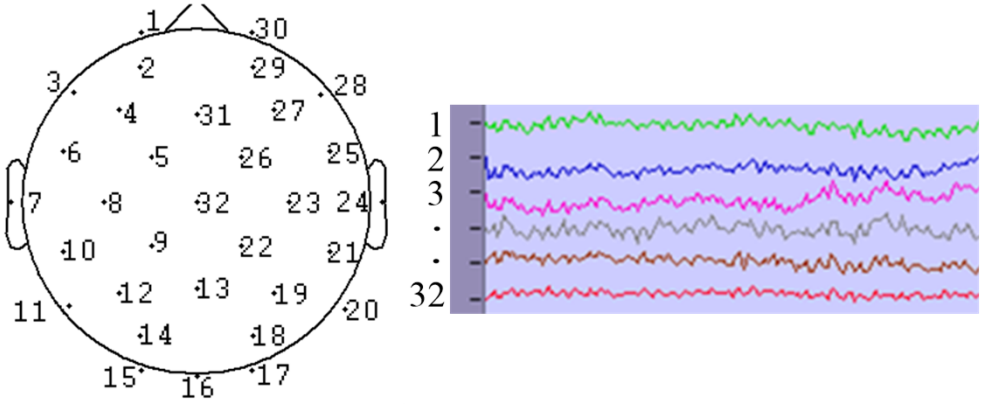
- Digital data consists exclusively of 0s and 1s An example of a digital sensor is an accelerometer, which sends a series of data points (speed, direction, and so on) to the Arduino. Usually digital sensors need a chip in the sensor to interpret the physical data.
- Because digital sensors send only data, they need a microcontroller to interpret the readings. A digital light sensor, therefore, could not be used as a photo resistor, although you could rig a microcontroller-controlled circuit to do the same thing.
- Analog data is transmitted as a continuous signal, almost like a wave. In other words, an analog sensor doesn't send a burst of 1s and 0s like digital sensors do; instead, the sensor modulates a continuous signal to transmit data.

1. Digital vs Analog

- You can use analog sensors in circuits without needing a microcontroller to interpret the signal. In the case of the photo resistor, you could actually use an analog light sensor as a photo resistor.



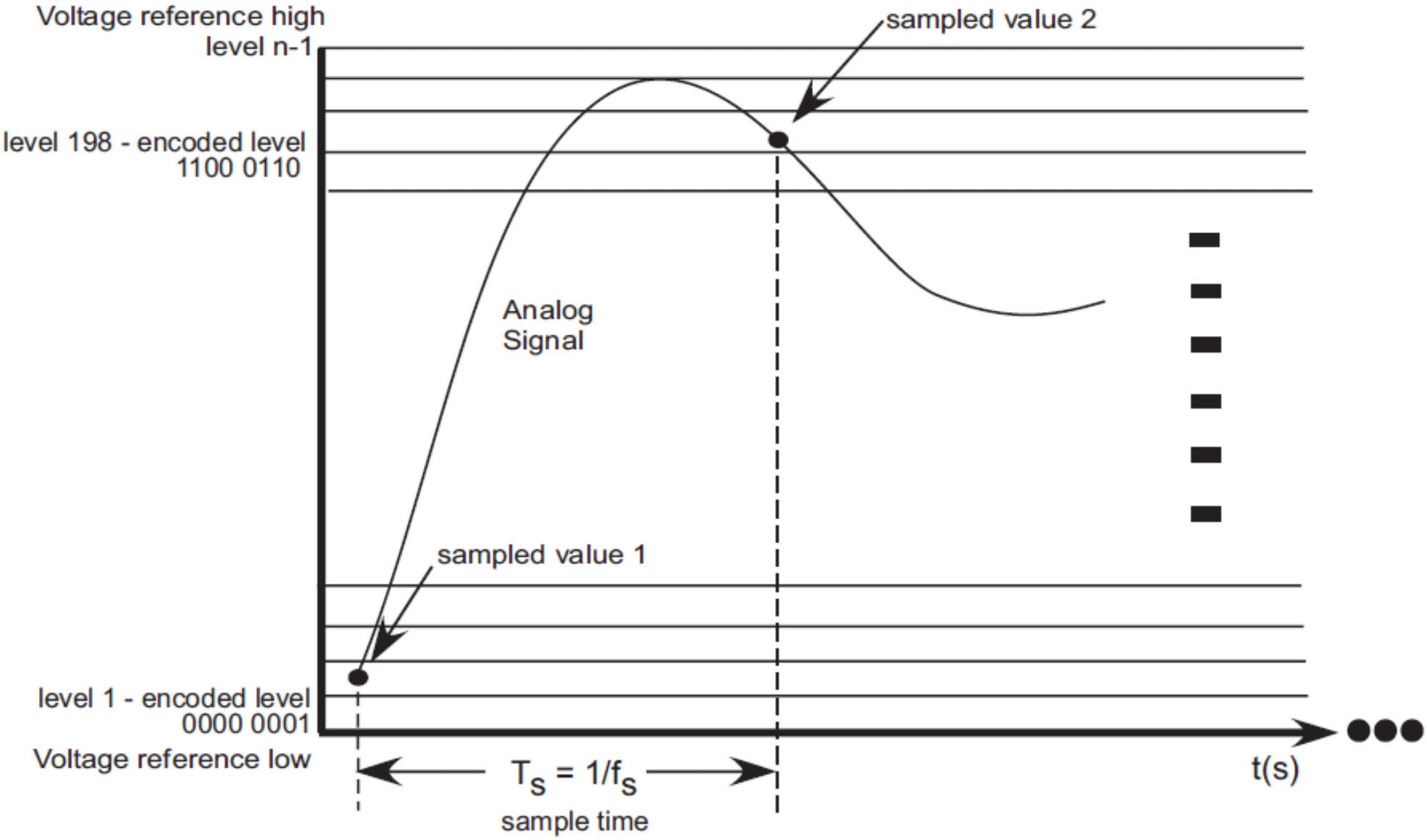
- Example of brain signal (analog)



2. ADC Theory

- A microcontroller is used to process information from the natural world, decide on a course of action based on the information collected, and then issue control signals to implement the decision.
- Since the information from the natural world, is analog or continuous in nature, and the microcontroller is a digital or discrete based processor, a method to convert an analog signal to a digital form is required.
- An analog to digital (ADC) system performs this task while a digital to analog converter (DAC) performs the conversion in the opposite direction.
- Most microcontrollers are equipped with an ADC subsystem; whereas, DACs must be added as an external peripheral device to the controller.

2. ADC Theory: Sampling, Quantization and Encoding



2. ADC Theory: Sampling, Quantization and Encoding

- **Sampling** is the process of taking ‘snap shots’ of a signal over time. When we sample a signal, we want to sample it in an optimal fashion such that we can capture the essence of the signal while minimizing the use of resources.
- In essence, we want to minimize the number of samples while retaining the capability to faithfully reconstruct the original signal from the samples. Intuitively, the rate of change in a signal determines the number of samples required to faithfully reconstruct the signal, provided that all adjacent samples are captured with the same sample timing intervals.
- Sampling is important since when we want to represent an analog signal in a digital system, such as a computer, we must use the appropriate sampling rate to capture the analog signal for a faithful representation in digital systems.

2. ADC Theory: Sampling, Quantization and Encoding

- Harry Nyquist from Bell Laboratory studied the sampling process and derived a criterion that determines the minimum sampling rate for any continuous analog signals. His, now famous, minimum sampling rate is known as the Nyquist sampling rate, which states that one must sample a signal at least twice as fast as the highest frequency content of the signal of interest.
- For example, if we are dealing with the human voice signal that contains frequency components that span from about 20 Hz to 4 kHz, the Nyquist sample theorem requires that we must sample the signal at least at 8 kHz, 8000 ‘snap shots’ every second.
- A low pass anti-aliasing filter must be employed to insure the Nyquist sampling rate is not violated. In the example above, a low pass filter with a cut off frequency of 4 KHz would be used before the sampling.

2. ADC Theory: Sampling, Quantization and Encoding

- **Quantization:** each digital system has a number of bits it uses as the basic unit to represent data. A bit is the most basic unit where single binary information, one or zero, is represented.
- The quantization of a sampled signal is how the signal is represented as one of the quantization levels. Suppose you have a single bit to represent an incoming signal. You only have two different numbers, 0 and 1. You may say that you can distinguish only low from high. Suppose you have two bits. You can represent four different levels, 00, 01, 10, and 11.What if you have three bits? You now can represent eight different levels: 000, 001, 010, 011, 100, 101, 110, and 111.

2. ADC Theory: Sampling, Quantization and Encoding

- When you had two bits, you were able to represent four different levels. If we add one more bit, that bit can be one or zero, making the total possibilities eight. Similar discussion can lead us to conclude that given n bits, we have 2^n unique numbers or levels one can represent.
- In many digital systems, the incoming signals are voltage signals. The voltage signals are first obtained from physical signals (pressure, temperature, etc.) with the help of transducers, such as microphones, angle sensors, and infrared sensors. The voltage signals are then conditioned to map their range with the input range of a digital system, typically 0 to 5 volts.

2. ADC Theory: Sampling, Quantization and Encoding

- n bits allow you to divide the input signal range of a digital system into 2^n different quantization levels. As can be seen from the figure, the more quantization levels means the better mapping of an incoming signal to its true value. If we only had a single bit, we can only represent level 0 and level 1.
- Any analog signal value in between the range had to be mapped either as level 0 or level 1, not many choices. Now imagine what happens as we increase the number of bits available for the quantization levels.
- What happens when the available number of bits is 8? How many different quantization levels are available now? Yes, 256. How about 10, 12, or 14? Notice also that as the number of bits used for the quantization levels increases for a given input range the 'distance' between two adjacent levels decreases accordingly.

2. ADC Theory: Sampling, Quantization and Encoding

- **Encoding** process involves converting a quantized signal into a digital binary. The quantization levels are determined by the eight bits and each sampled signal is quantized as one of 256 quantization levels. Consider the two sampled signals, rhe first sample is mapped to quantization level 2 and the second one is mapped to quantization level 198. Note the amount of quantization error introduced for both samples. The quantization error is inversely proportional to the number of bits used to quantize the signal.
- Once a sampled signal is quantized, the encoding process involves representing the quantization level with the available bits. Thus, for the first sample, the encoded sampled value is 0000_0001, while the encoded sampled value for the second sample is 1100_0110. As a result of he encoding process, sampled analog signals are now represented as a set of binary numbers. Thus, the encoding is the last necessary step to represent a sampled analog signal into its corresponding digital form.

2. ADC Theory: Resolution

- **Resolution** is a measure used to quantize an analog signal. In fact, resolution is nothing more than the voltage 'distance' between two adjacent quantization levels we discussed earlier.
- Suppose again we have a range of 5 volts and one bit to represent an analog signal. The resolution in this case is 2.5 volts, a very poor resolution. You can imagine how your TV screen will look if you only had only two levels to represent each pixel, black and white. The maximum error, called the resolution error, is 2.5 volts for the current case, 50 % of the total range of the input signal.

2. ADC Theory: Resolution

- Suppose you now have four bits to represent quantization levels. The resolution now becomes 1.25 volts or 25% of the input range. Suppose you have 20 bits for quantization levels. The resolution now becomes 4.77×10^{-6} volts, 9.54×10^{-5}% of the total range.
- The number of bits used for the quantization is directly proportional to the resolution of a system. In general, resolution may be defined as:

$$resolution = (voltage\ span)/2^b = (V_{ref\ high} - V_{ref\ low})/2^b$$

- for the Arduino Mega, the resolution is:

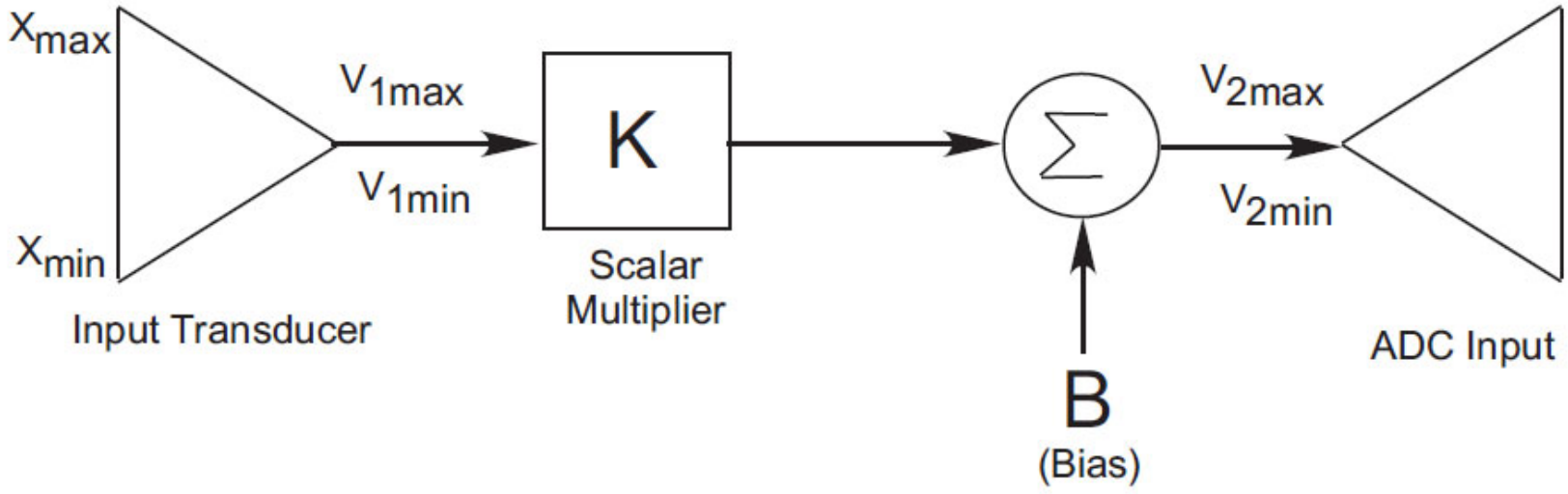
$$resolution = (5 - 0)/2^{10} = 4.88\,mV$$

2. ADC Theory: Data rate and Dynamic Range

- **Data rate** is the amount of data generated by a system per some time unit. Typically, the number of bits or the number of bytes per second is used as the data rate of a system.
- **Dynamic range** is a measure used to describe the signal to noise ratio. The unit used for the measurement is Decibel (dB), which is the strength of a signal with respect to a reference signal.The greater the dB number, the stronger the signal is compared to a noise signal.
- The definition of the dynamic range is $20\ log\ 2^b$ where b is the number of bits used to convert analog signals to digital signals. Typically, you will find 8 to 12 bits used in commercial ADC converters, translating the dynamic range from $20\ log\ 2^8$ dB to $20\ log\ 2^{12}$ dB.

2. ADC Theory: ADC Process

- A block diagram of the signal conditioning for an analog-to-digital converter. The range of the sensor voltage output is mapped to the analog-to-digital converter input: voltage range. The scalar multiplier maps the magnitudes of the two ranges and the bias voltage is used to align two limits.

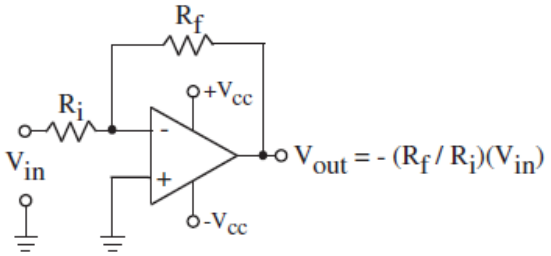


2. ADC Theory: ADC Process

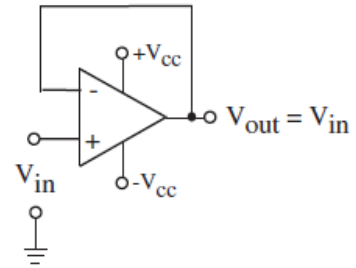
- We also need a signal conditioning circuitry before we apply the ADC. The signal conditioning circuitry is called the transducer interface. The objective of the transducer interface circuit is to scale and shift the electrical signal range to map the output of the input transducer to the input range of the analog-to-digital converter which is typically 0 to 5 VDC.
- Operational amplifiers (op amps) are typically used to implement a transducer interface design.

2. ADC Theory: ADC Process

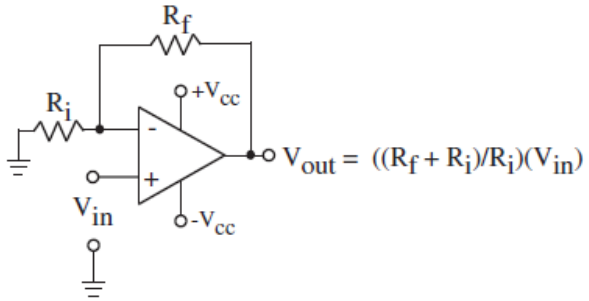
- Op-amp configuration



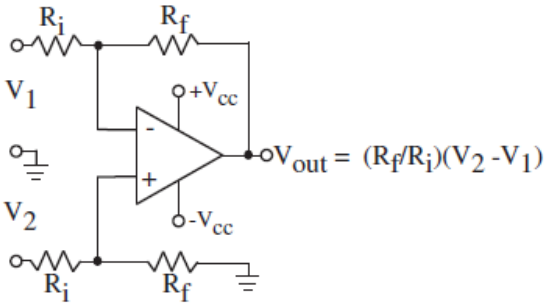
a) Inverting amplifier



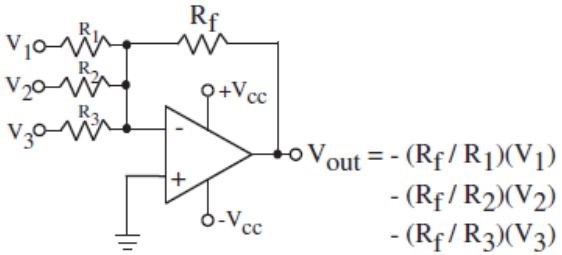
b) Voltage follower



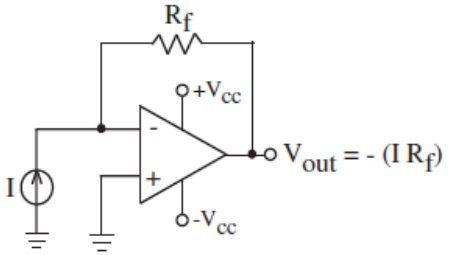
c) Non-inverting amplifier



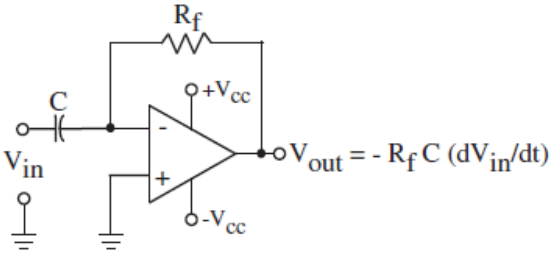
d) Differential input amplifier



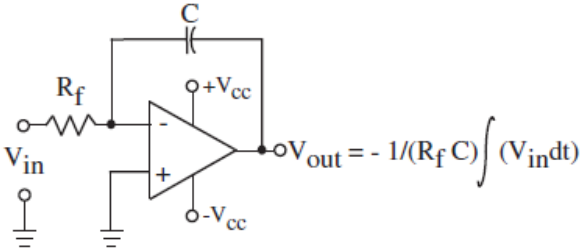
e) Scaling adder amplifier



f) Transimpedance amplifier
(current-to-voltage converter)



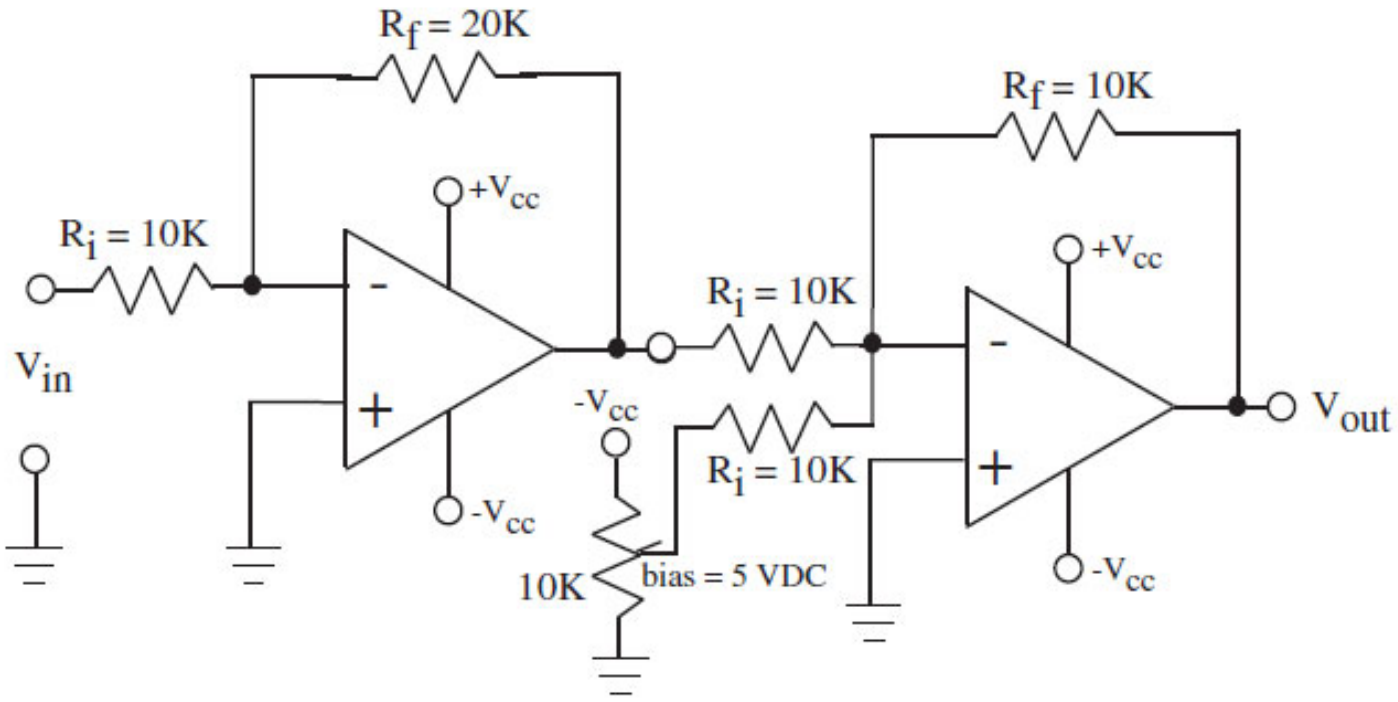
g) Differentiator



h) Integrator

2. ADC Theory: ADC Process

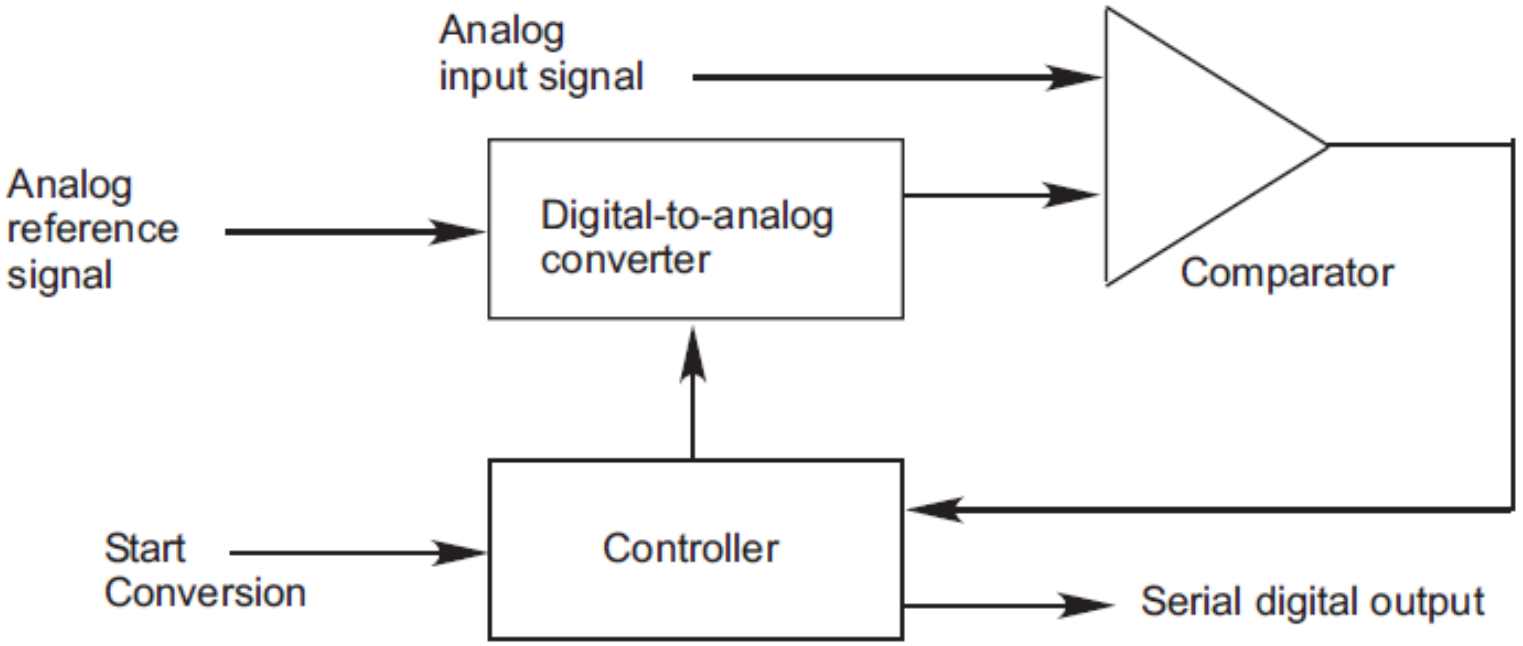
- Operational amplifier implementation of the transducer interface design (TID) example circuit.



2. ADC Theory: ADC Process

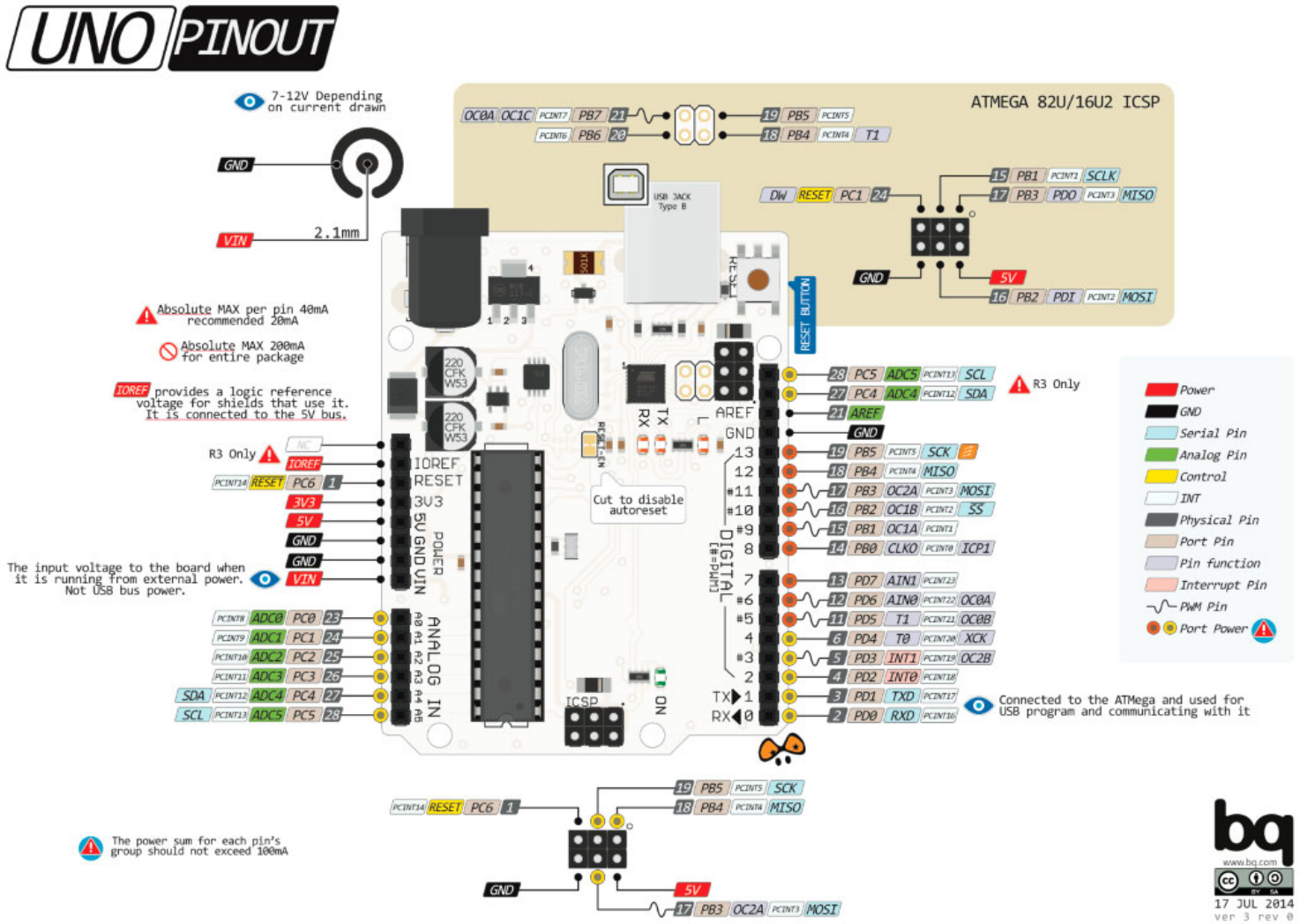
- Arduino Uno/Mega is equipped with a successive-approximation ADC converter.
- The successive-approximation technique uses a digital-to-analog converter, a controller,and a comparator to perform the ADC process. Starting from the most significant bit down to the least significant bit, the controller turns on each bit at a time and generates an analog signal, with the help of the digital-to-analog converter, to be compared with the original input analog signal.
- Based on the result of the comparison, the controller changes or leaves the current bit and turns on the next most significant bit. The process continues until decisions are made for all available bits. The advantage of this technique is that the conversion time is uniform for any input, but the disadvantage of the technology is the use of complex hardware for implementation.

2. ADC Theory: ADC Process



Successive-approximation converter

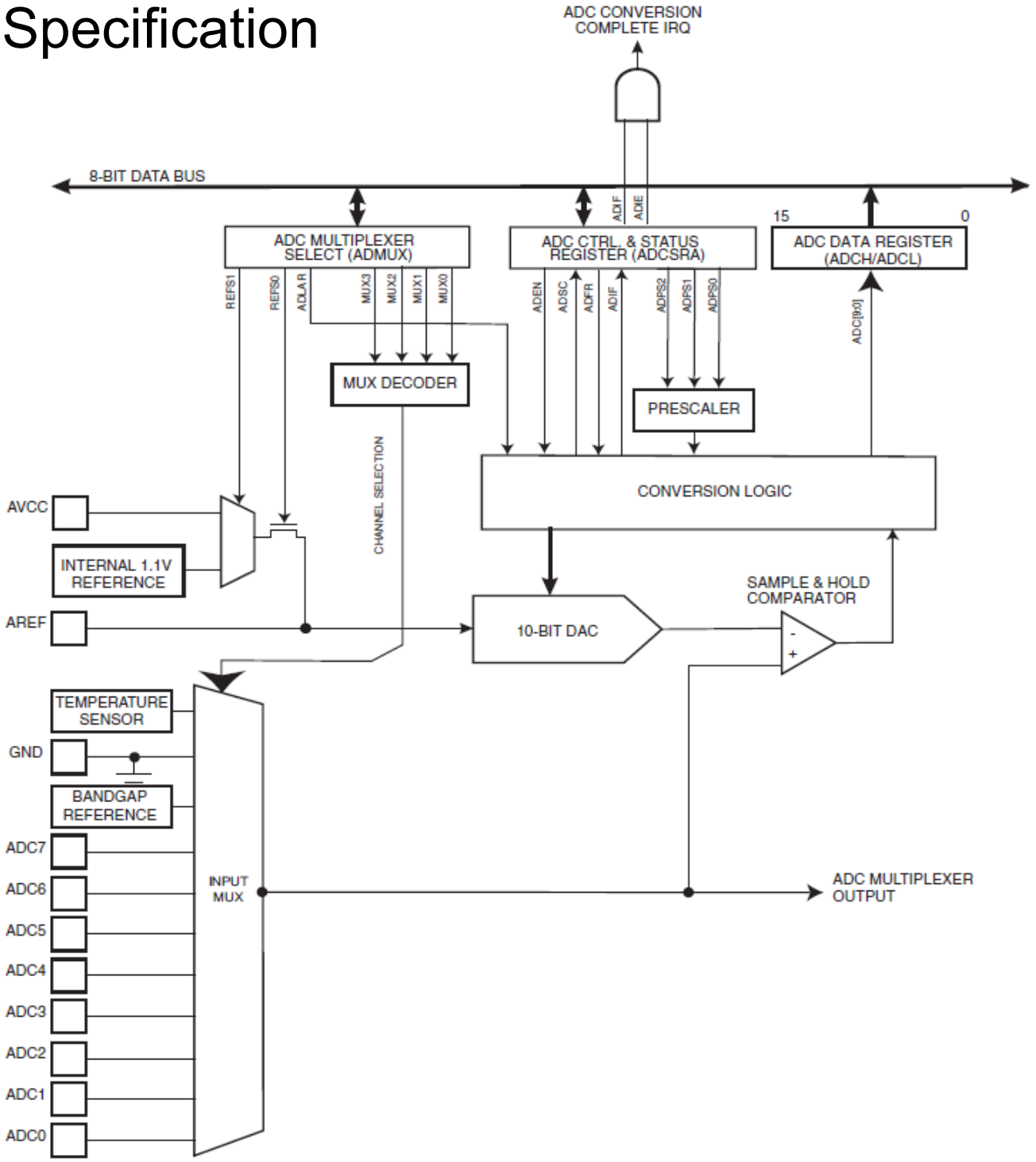
3. Arduino – ADC Specification



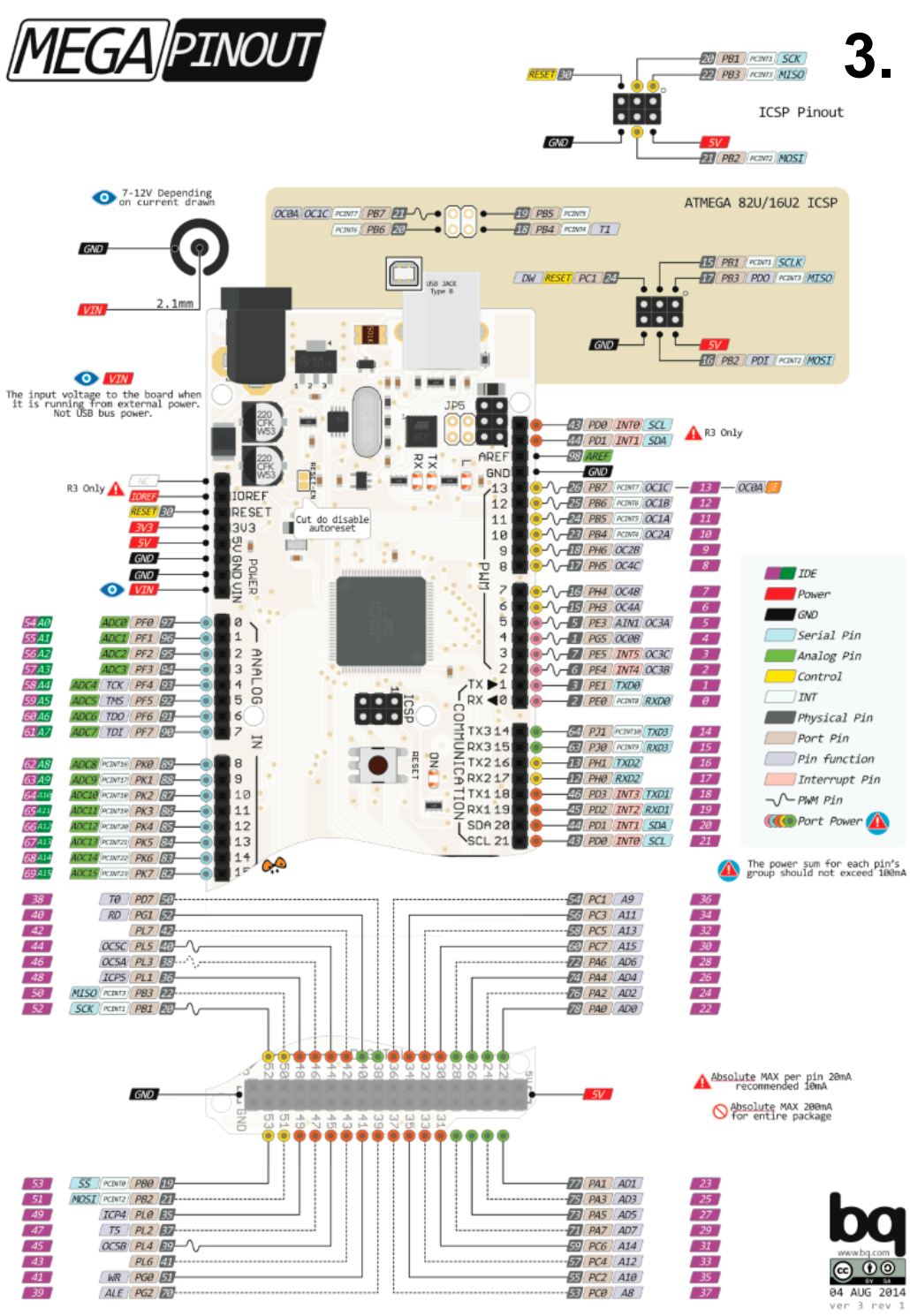
- ATmega328P
- 16 MHz Clock
- 32 KB Flash Memory
- 2 KB SRAM
- 14 Digital I/O Pins
- **6 Analog input Pins, a 10-bit** resolution on each pin.

3. Arduino – ADC Specification

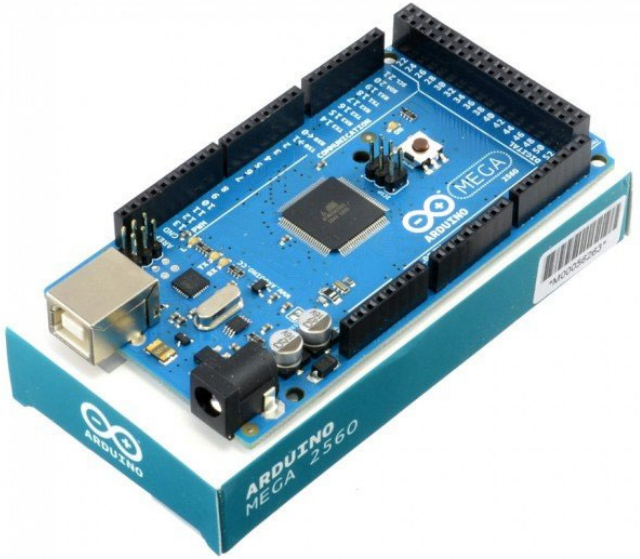
- Arduino Uno –ADC
- ATmega328P:
  - 8-channel 10-bit ADC in TQFP and QFN/MLF package
  - **6-channel 10-bit ADC in PDIP Package**



3. Arduino – ADC Specification

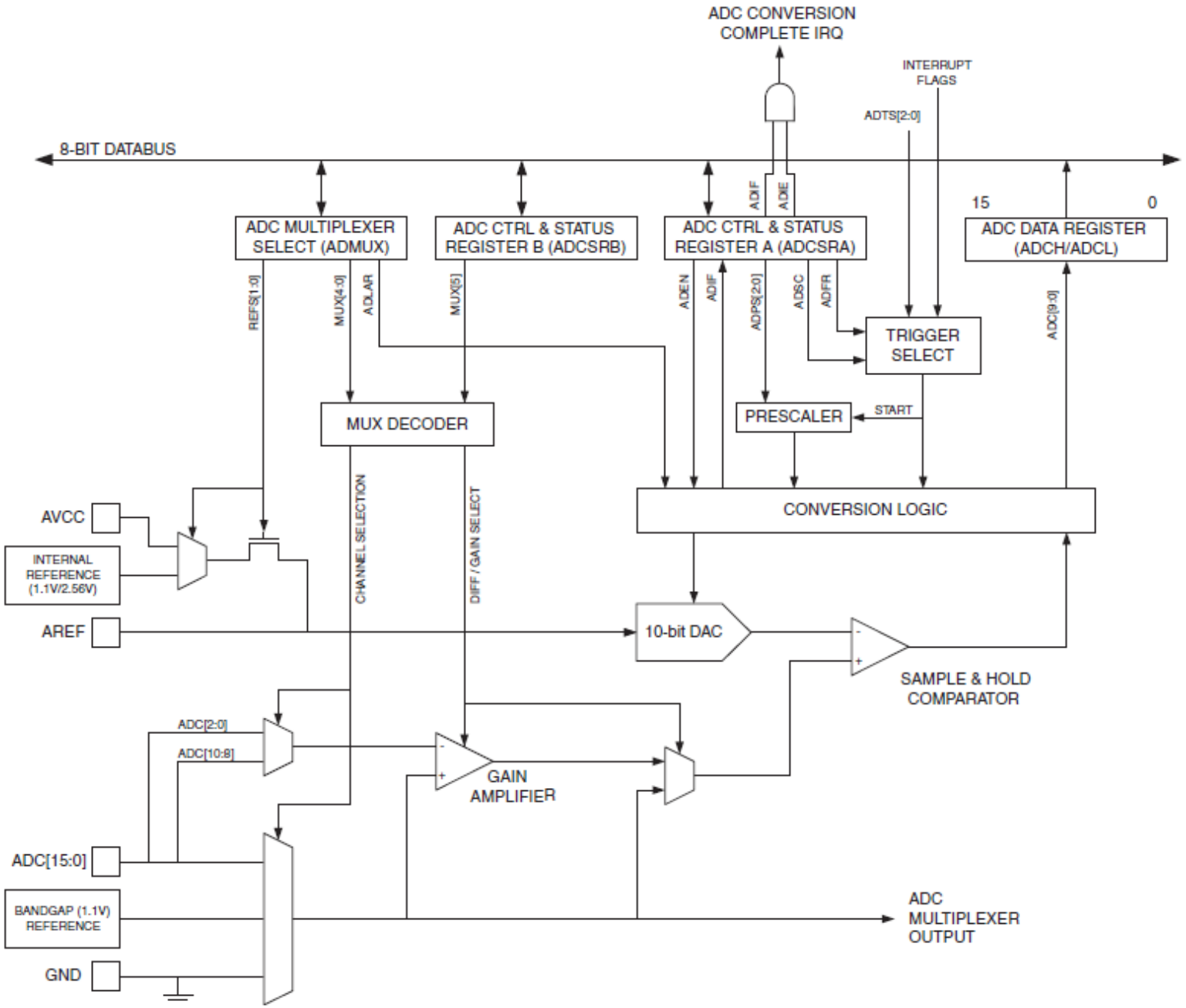


- ATmega2560
- 16MHz Clock
- 256 KB Flash Memory
- 8 KB SRAM
- 54 Digital I/O Pins, 15 of these can be used with PWM.
- **16 Analog Input Pins**

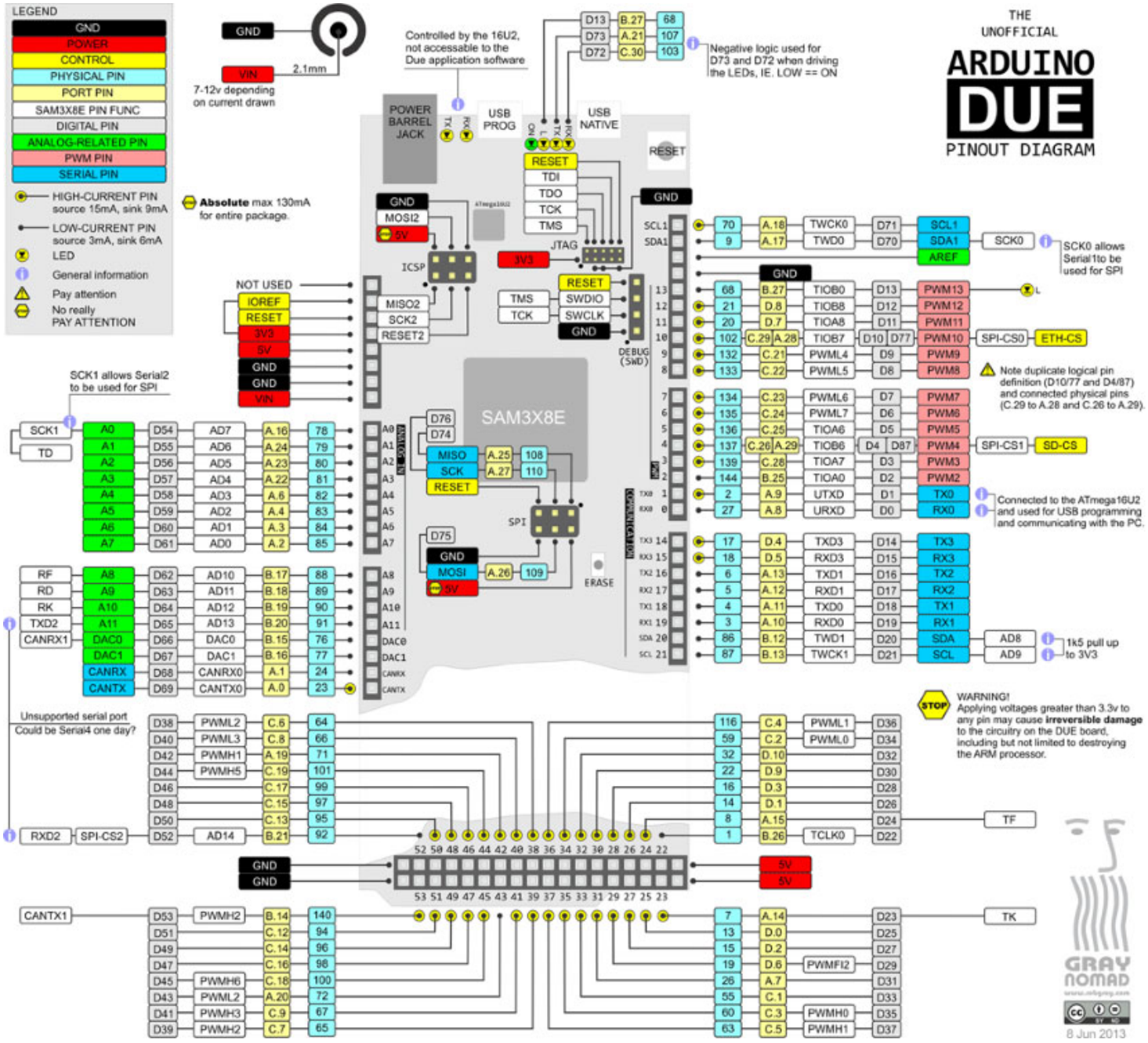


3. Arduino – ADC Specification

- Arduino Mega ADC



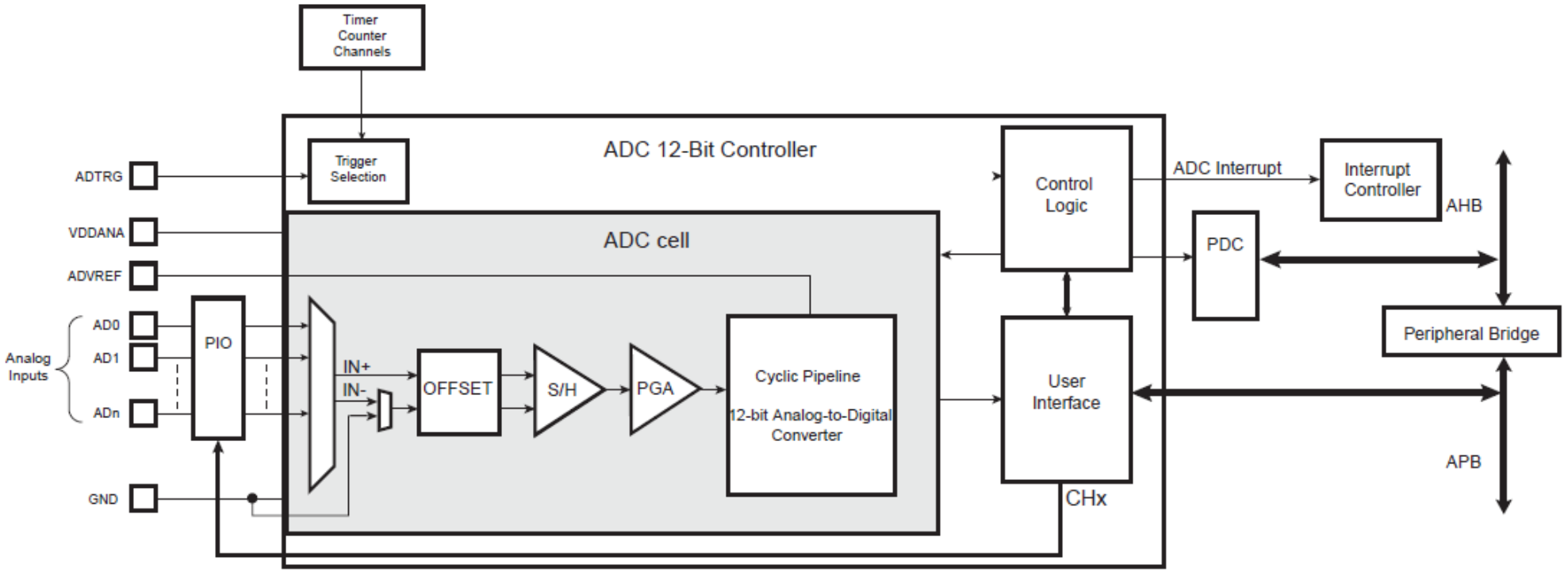
3. Arduino – ADC Specification



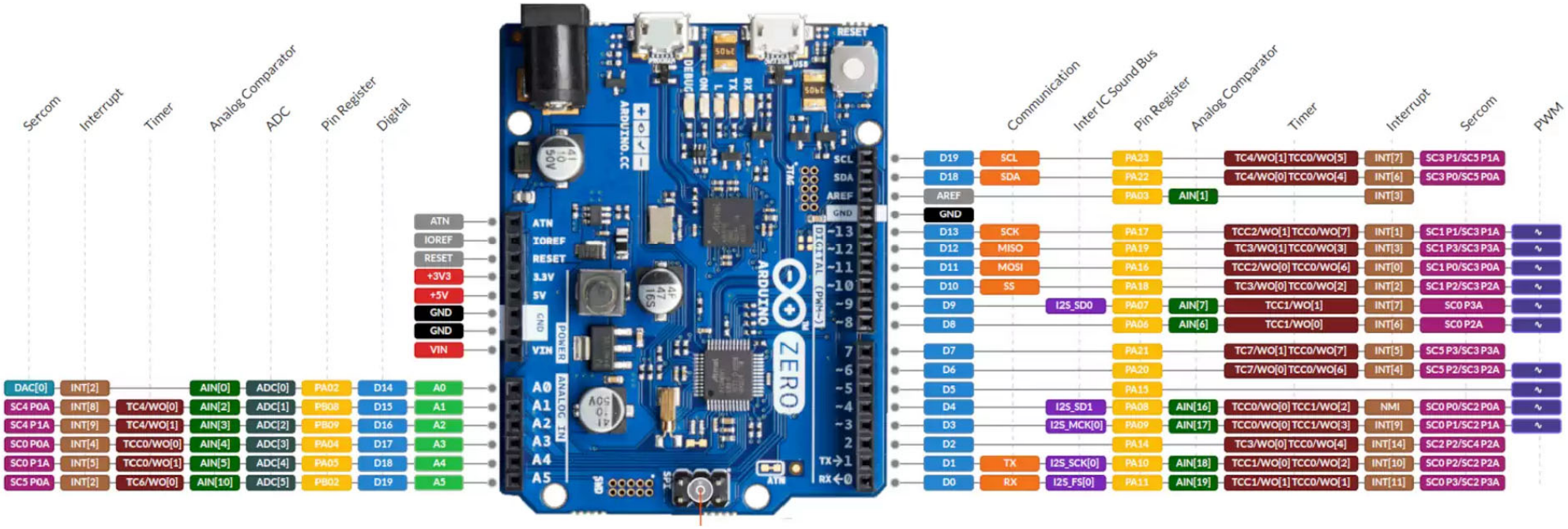
- ❑ AT91SAM3X8E
- ❑ 84 MHz Clock
- ❑ 512 KB Flash Memory
- ❑ 96 KB SRAM
- ❑ 54 Digital I/O Pins
- ❑ **12 Analog Input pins**
- ❑ 2 Analog Output Pins, DAC1 & DAC2 pins use a DAC to provide 12-bit Analog outputs.

3. Arduino – ADC Specification

- Arduino Due -ADC



3. Arduino – ADC Specification

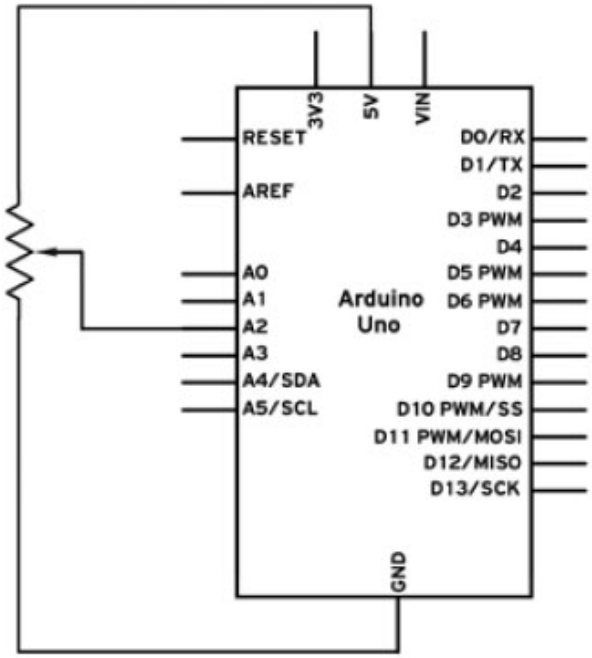


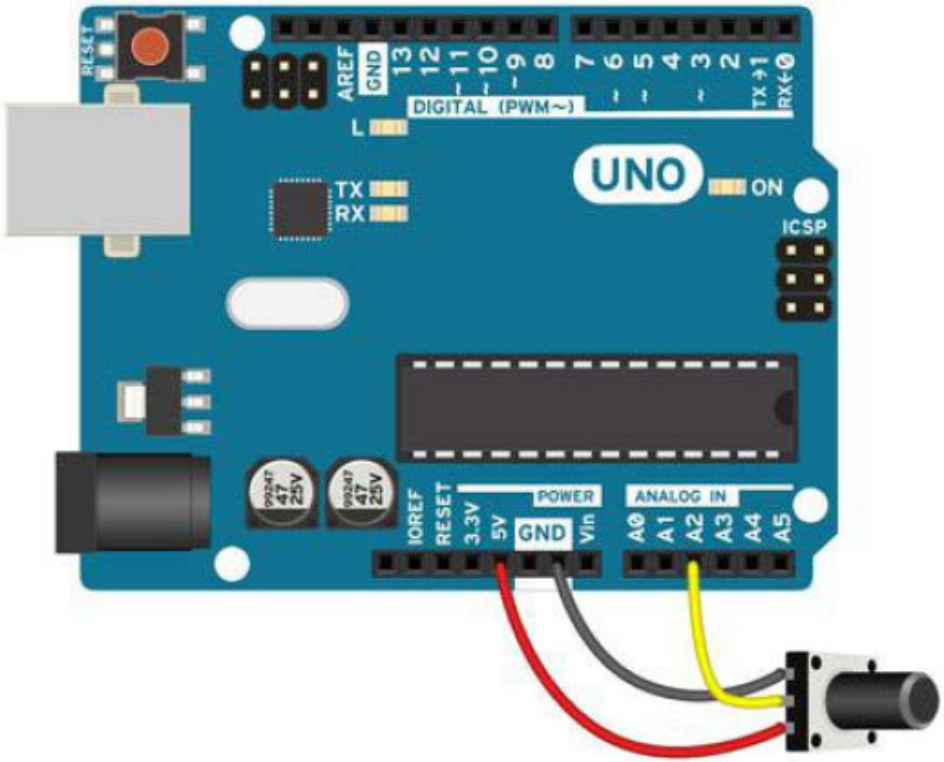
- ❑ ATSAMD21G18, 32-Bit ARM® Cortex® M0+
- ❑ 48 MHz Clock
- ❑ 256 KB Flash Memory
- ❑ 32 KB SRAM
- ❑ 20 Digital I/O Pins
- ❑ **6 Analog Input pins, 12-bit ADC channels**
- ❑ 1 Analog Output Pins, 10 bit DAC

4. Potentiometer

- A potentiometer, also called a variable resistor, is a basic component that allows us to modify its internal resistance. We can use it to adjust settings in our program at any time. Or, we can use them to control things, such as a robotic hand or the intensity of a particular light.
- The following example, we will make the built-in LED blink with a frequency that we will control via the potentiometer and also print the values over serial.
- Wiring: The potentiometer has three terminals. Connect the terminal in the center to an analog pin; here, we will connect it to A2. Connect one of the other terminals to GND. Connect the third terminal to 5V.

4. Potentiometer





4. Potentiometer

- The following code will read the value of the potentiometer, print it on the serial connection, and vary the LED pulsing frequency accordingly:

```
int LED = 13;       // Declare the built-in LED
int sensorPin = A2; // Declare the analog port we connected

void setup(){
  // Start the Serial connection
  Serial.begin(9600);
  // Set the built in LED pin as OUTPUT
  pinMode(LED, OUTPUT);
}


void loop(){
  // Read the value of the sensor
  int val = analogRead(sensorPin);
  // Print it to the Serial
  Serial.println(val);

  // Blink the LED with a delay of a forth of the sensor value
  digitalWrite(LED, HIGH);
  delay(val/4);
  digitalWrite(LED, LOW);
  delay(val/4);
}
```

4. Potentiometer

- Inside each Arduino there is an Analog-to-Digital Converter (ADC). This component can convert an analog signal value to a digital representation. ADCs come in a variety of ranges, accuracies, and resolutions. The integrated models from the Uno, Mega and other normal Arduinos have a 10-bit resolution.
- This means that a voltage between 0 and 5 V on 5V Arduinos will be represented by a corresponding value between 0 and 1023. A voltage of 2.5 V will be equal to 512, which is half of the range.
- We should never exceed the maximum voltage of the board on the analog inputs. In most boards, this is 5 V, but on the Due and a few others, the voltage can be 3.3 V. A potentiometer works by adjusting the conductor length between the central and side erminals. It is recommended to use a high-resistance potentiometer; otherwise, a lot of current will pass through, heating it up. Any value over 10K ohm should be good.

4. Potentiometer

- First, we declare two variables for the built-in LED and for the used analog port, to which we connected the potentiometer:

```
int LED = 13;
int sensorPin = A2;
```

- In the setup() function, we start the serial connection and we declare the LED pin as an output:

```
void setup(){
  Serial.begin(9600);
  pinMode(LED, OUTPUT);
}
```

- The deal breaker is the following function. It reads the analog value of the specified analog input and it returns it as a number between 0 and 1023. Remember that this conversion takes around 100 microseconds on most Arduino boards.

```
int val = analogRead(sensorPin);
```

4. Potentiometer

- And now we do two things. We print the value on the serial connection, and then we make the LED blink with an in-between delay of the read value divided by four, to make it blink fast:

```
Serial.println(val);
digitalWrite(LED, HIGH);
delay(val/4);
digitalWrite(LED, LOW);
delay(val/4);
```

- The analogRead() function is one of the most important functions on the Arduino platform. Almost every sensor uses this kind of interfacing. There are a few more things to know.

4. Potentiometer

- The **Arduino Due** has a few great features on the analog side. First of all, it has an integrated 12-bit ADC, so it can return more precise values between 0 and 4095.
- However, it comes preconfigured to only output 10 bit. We can change that using the analogReadResolution(bits) function. For example, analogReadResolution(12) will make the analogRead() function output 12-bit values.
- Remember that the Due is designed for a maximum of 3.3V, not 5 V; applying more than 3.3 V will damage the board.
- **Analog reference (AREF)**: Most Arduinos have an AREF pin that enables us to give the voltage range on which the ADC will return. So if we input 2 V to the AREF pin and configure the code, it will output 1023 for 2V and 0 for 0V. This feature is useful if we have sensors that output less than 5V and we need more precision.

4. Potentiometer

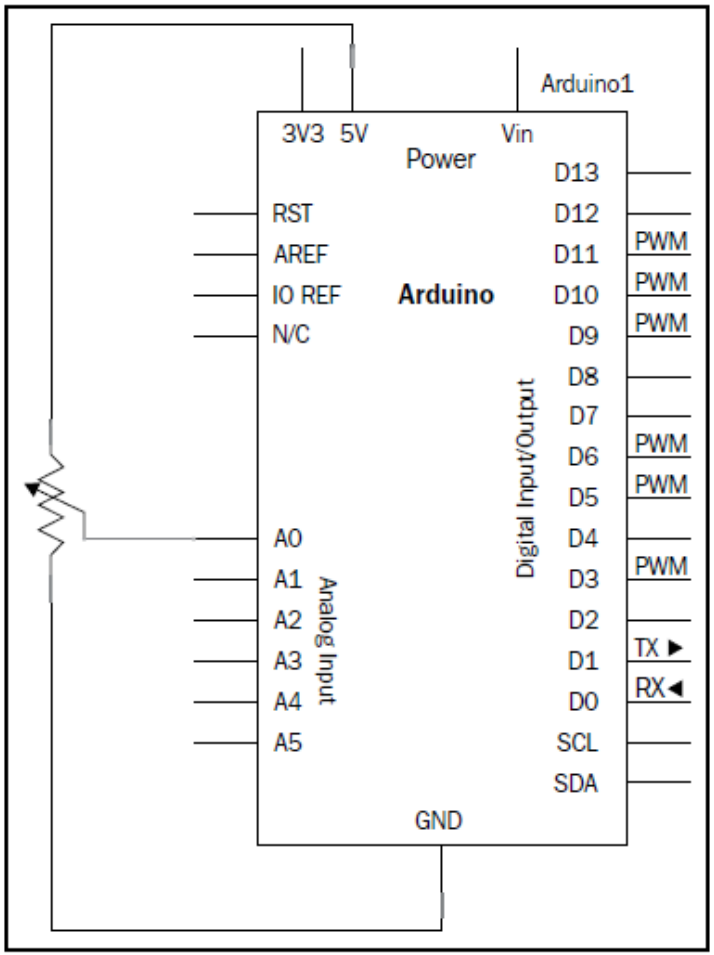
- To tell the Arduino we are using an external reference on AREF, we need to use the analogReference(type) function. The type argument can take the following values:
  - DEFAULT: This is the standard configuration with a range from 0 V to 5 V
  - EXTERNAL: This will use the value on AREF for reference
- Another important thing to remember is to use the analogReference() function first, before using the analogRead() function. If the reference type is not set to EXTERNAL, but we do apply a voltage to AREF, when we use the analogRead() function we will basically short the microcontroller. This can damage the board.
- For more information on the Arduino Due analogReadResolution() function and more analog references, visit the following links:
  - http://arduino.cc/en/Reference/AnalogReadResolution
  - http://arduino.cc/en/Reference/AnalogReference

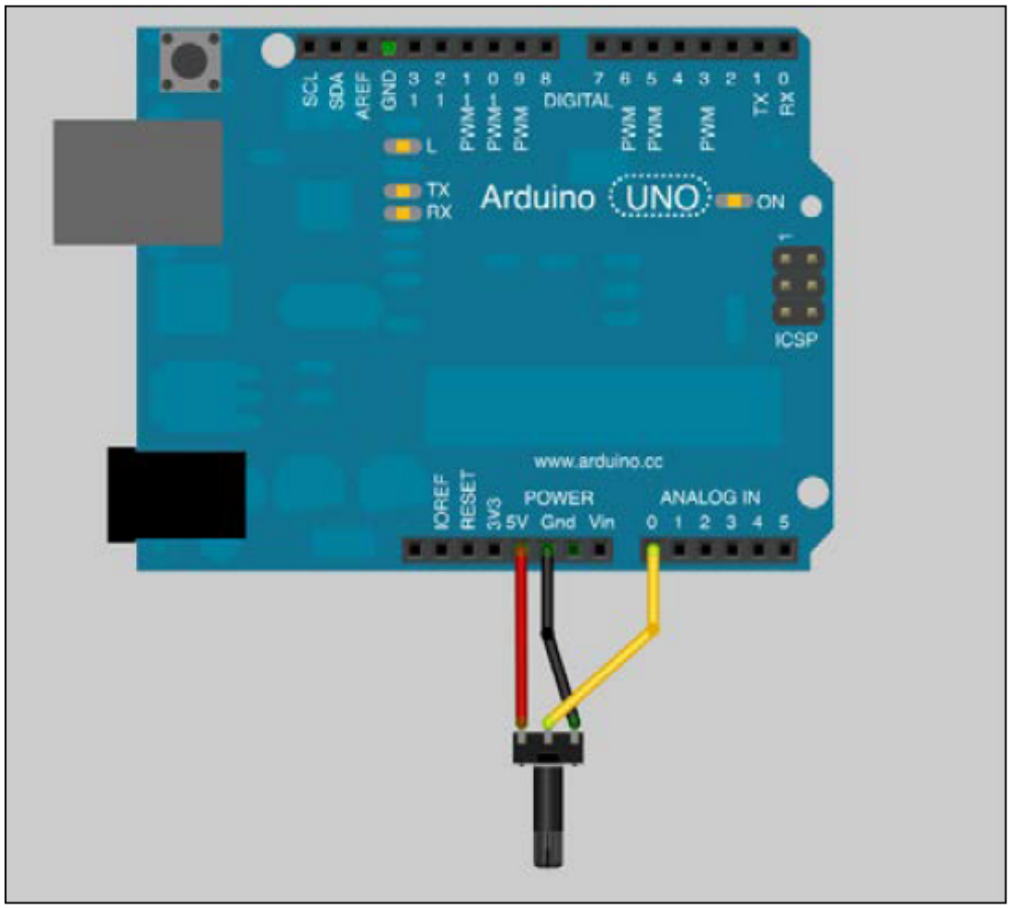
4. Potentiometer - low voltage voltmeter

- Measuring voltage requires two different points on a circuit. Indeed, a voltage is an electrical potential. Here, we have (only) that analog pin involved in our circuit to measure voltage.
- We're using the +5 V supply from Vcc as a reference. We control the resistance provided by the potentiometer and supply the voltage from the Vcc pin to have something to demonstrate.
- If we want to use it as a real potentiometer, we have to supply another part of a circuit with Vcc too, and then connect our A0 pin to another point of the circuit.
- As we saw, the analogRead() function only provides integers from 0 to 1023.
- The range 0 to 1023 is mapped to 0 to 5V. That comes built into the Arduino. We can then calculate the voltage as follows:

  V = 5 * (analogRead() value / 1023)

4. Potentiometer - low voltage voltmeter





4. Potentiometer - low voltage voltmeter

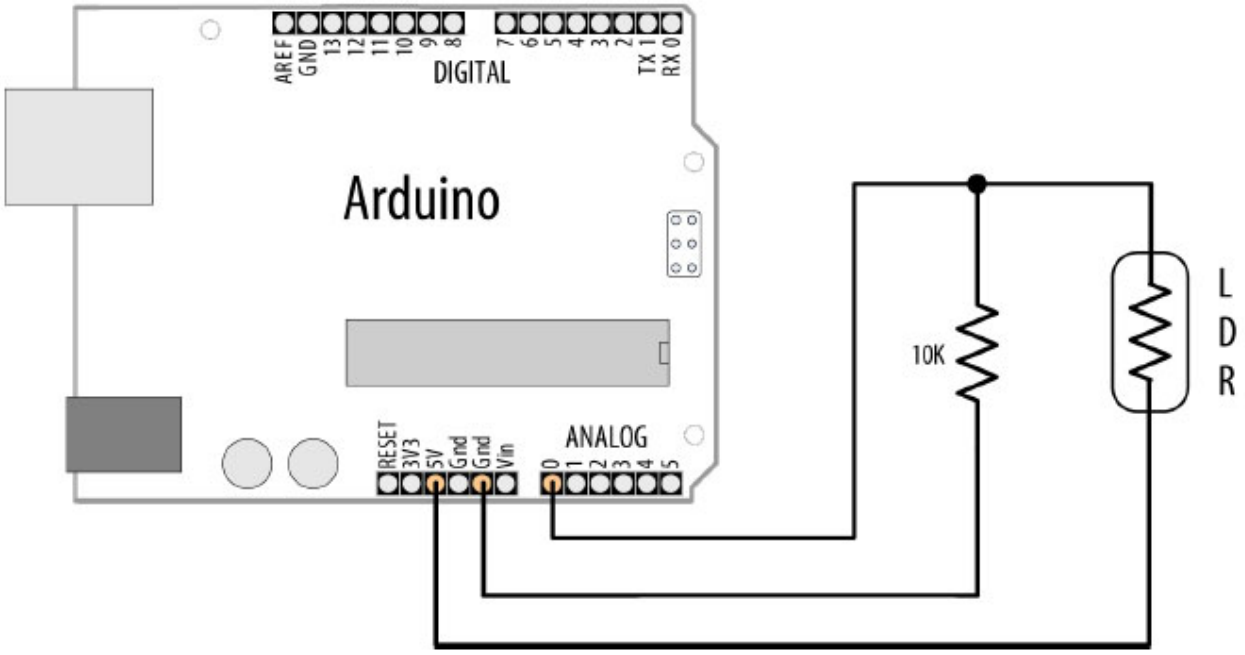
```
int potPin = 0;      // pin number where the potentiometer is connected
int ledPin = 13 ;    // pin number of the on-board LED
int potValue = 0 ;   // variable storing the voltage value measured at
potPin pin
float voltageValue = 0.; // variable storing the voltage calculated

void setup() {
  Serial.begin(9600);
  pinMode(ledPin, OUTPUT);  // define ledPin pin as an output
}

void loop(){
  potValue = analogRead(potPin); // read and store the read value at
potPin pin

  digitalWrite(ledPin, HIGH);    // turn on the LED
  delay(potValue);               // pause the program during potValue
millisecond
  digitalWrite(ledPin, LOW);     // turn off the LED
  delay(potValue);               // pause the program during potValue
millisecond

  voltageValue = 5. * (potValue / 1023.) ;  // calculate the voltage

  Serial.println(voltageValue); // write the voltage value an a
carriage return
}
```

4. Potentiometer - low voltage voltmeter

- The serial communication and Serial.println() to write the current calculated voltage value to the serial communication port, followed by a carriage return. In order to see a result on your computer, you have to turn on the serial monitor, of course. Then, you can read the voltage values.
- Please note that we are using an ADC here in order to convert an analog value to digital; then, we are making a small calculation on that digital value in order to have a voltage value. This is a very expensive method compared to a basic analog voltage controller.
- It means our precision depends on the ADC itself, which has a resolution of 10 bits. It means we can only have 1024 values between 0 V and 5 V. 5 divided by 1024 equals 0.00488, which is approximated. It basically means we won't be able to distinguish between values such as 2.01 V and 2.01487 V, for instance.

5. Light Sensor - LDR

- The easiest way to detect light levels is to use a light dependent resistor (LDR). This changes resistance with changing light levels, and when connected in the circuit shown the following figure, it produces a change in voltage that the Arduino analog input pins can



5. Light Sensor - LDR

```
const int ledPin =  13;      // LED connected to digital pin 13
const int sensorPin = 0;     // connect sensor to analog input 0

void setup()
{
  pinMode(ledPin, OUTPUT);  // enable output on the led pin
}

void loop()
{
  int rate = analogRead(sensorPin);    // read the analog input
  digitalWrite(ledPin, HIGH);   // set the LED on

  delay(rate);                  // wait duration dependent on light level
  digitalWrite(ledPin, LOW);    // set the LED off
  delay(rate);
}
```

5. Light Sensor - LDR

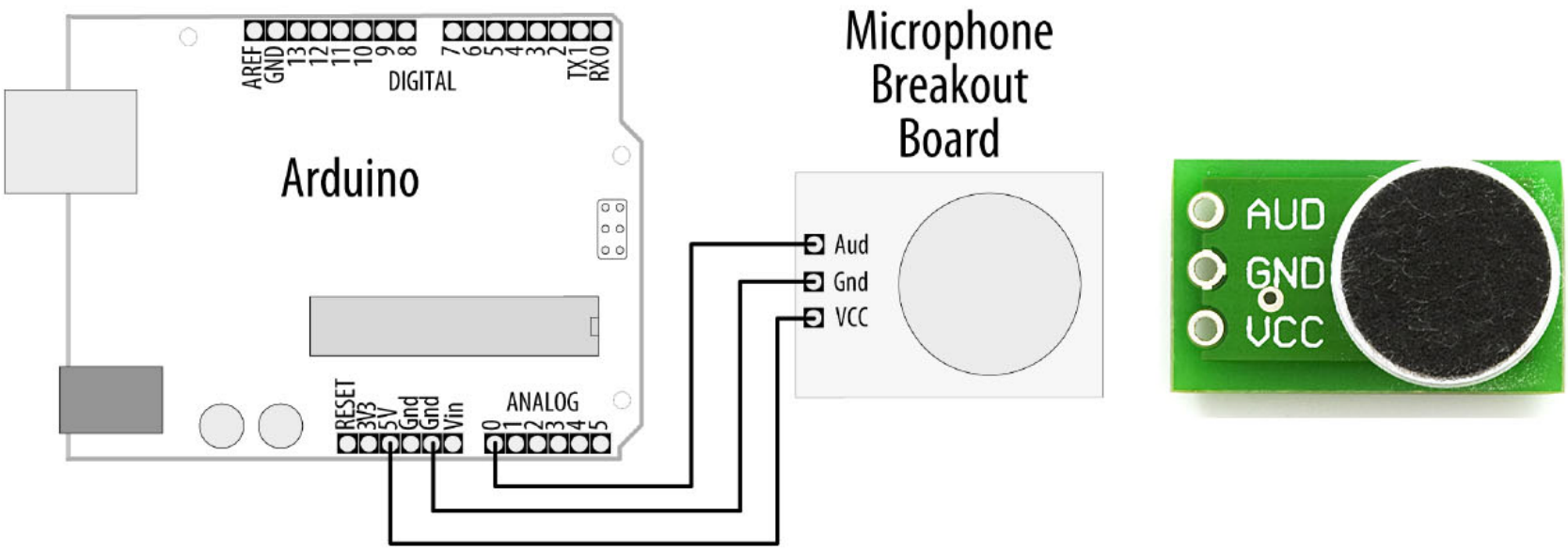
- The circuit for this recipe is the standard way to use any sensor that changes its resistance based on some physical phenomenon.
- The voltage on analog pin 0 changes as the resistance of the LDR changes with varying light levels.
- A circuit such as this will not give the full range of possible values from the analog input—0 to 1,023—as the voltage will not be swinging from 0 volts to 5 volts. This is because there will always be a voltage drop across each resistance, so the voltage where they meet will never reach the limits of the power supply. When using sensors such as these, it is important to check the actual values the device returns in the situation you will be using it. Then you have to determine how to convert them to the values you need to control whatever you are going to control.

5. Light Sensor - LDR

- The LDR is a simple kind of sensor called a resistive sensor. A range of resistive sensors respond to changes in different physical characteristics. Similar circuits will work for other kinds of simple resistive sensors, although you may need to adjust the resistor to suit the sensor.
- Choosing the best resistor value depends on the LDR you are using and the range of light levels you want to monitor. Engineers would use a light meter and consult the data sheet for the LDR, but if you have a multimeter, you can measure the resistance of the LDR at a light level that is approximately midway in the range of illumination you want to monitor. Note the reading and choose the nearest convenient resistor to this value.

6. Microphone- Detecting Sound

- If you want to detect sounds such as clapping, talking, or shouting, a microphone can be used, for example: using the BOB-08669 breakout board for the Electret Microphone



6. Microphone- Detecting Sound

- Coding: The built-in LED on Arduino pin 13 will turn on when you clap, shout, or play loud music near the microphone. You may need to adjust the threshold—use the Serial Monitor to view the high and low values, and change the threshold value so that it is between the high values you get when noise is present and the low values when there is little or no noise.

```
/*
microphone sketch

SparkFun breakout board for Electret Microphone is connected to analog pin 0
*/

const int ledPin = 13;            //the code will flash the LED in pin 13
const int middleValue = 512;      //the middle of the range of analog values
const int numberOfSamples = 128;  //how many readings will be taken each time

int sample;                       //the value read from microphone each time
long signal;                      //the reading once you have removed DC offset
long averageReading;              //the average of that loop of readings

long runningAverage=0;           //the running average of calculated values
const int averagedOver= 16;      //how quickly new values affect running average
                                 //bigger numbers mean slower

const int threshold=400;         //at what level the light turns on
```

6. Microphone- Detecting Sound

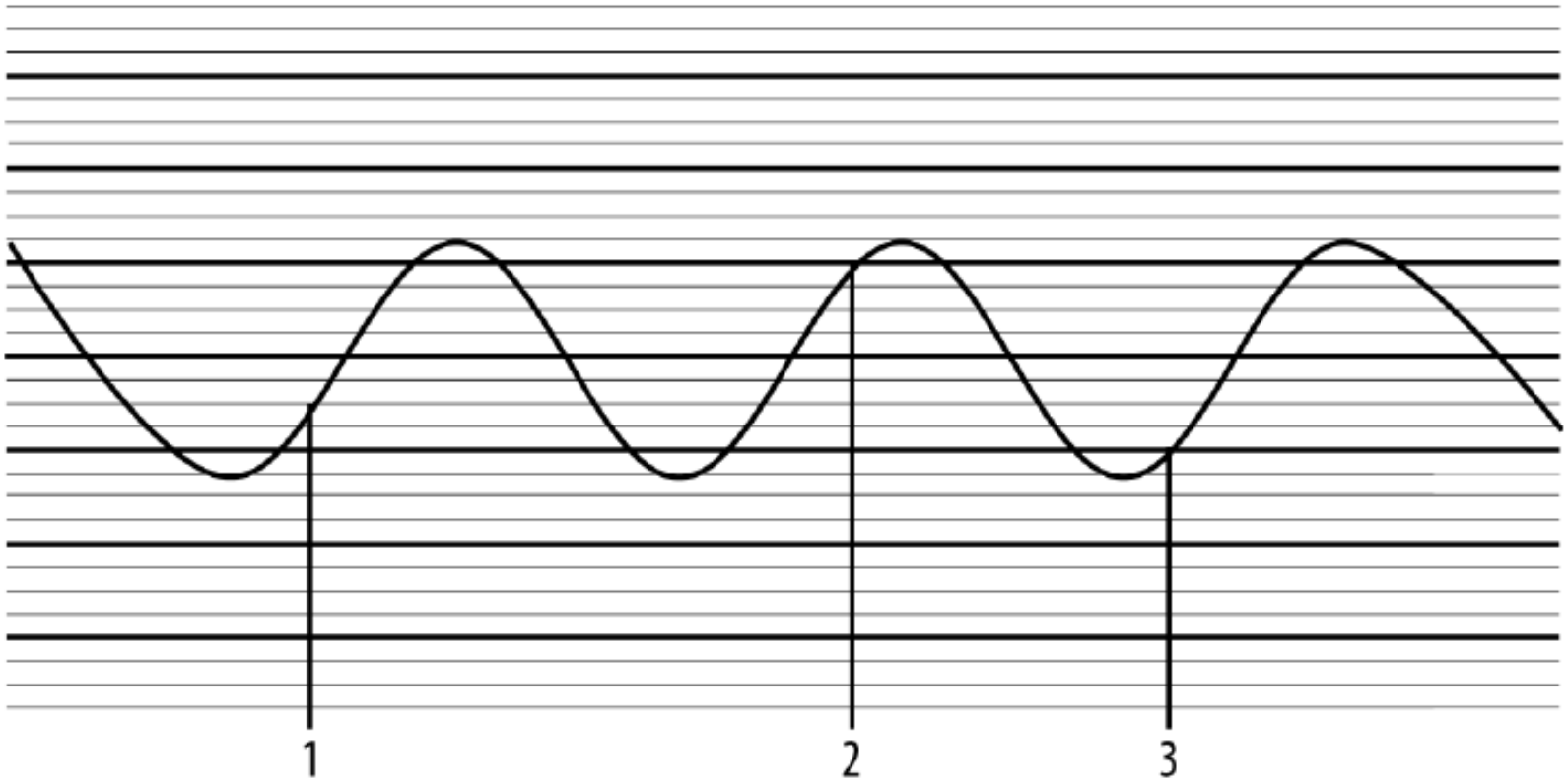
```
void setup() {
  pinMode(ledPin, OUTPUT);
  Serial.begin(9600);
}

void loop() {
  long sumOfSquares = 0;
  for (int i=0; i<numberOfSamples; i++) { //take many readings and average them
    sample = analogRead(0);                //take a reading
    signal = (sample - middleValue);       //work out its offset from the center
    signal *= signal;                      //square it to make all values positive
    sumOfSquares += signal;                //add to the total
  }
  averageReading = sumOfSquares/numberOfSamples;     //calculate running average
  runningAverage=(((averagedOver-1)*runningAverage)+averageReading)/averagedOver;

  if (runningAverage>threshold){          //is average more than the threshold ?
    digitalWrite(ledPin, HIGH);           //if it is turn on the LED
  }else{
    digitalWrite(ledPin, LOW);            //if it isn't turn the LED off
  }
  Serial.println(runningAverage);         //print the value so you can check it
}
```

6. Microphone- Detecting Sound

- A microphone produces very small electrical signals. If you connected it straight to the pin of an Arduino, you would not get any detectable change. The signal needs to be amplified first to make it usable by Arduino. The SparkFun board has the microphone with an amplifier circuit built in to amplify the signal to a level readable by Arduino.
- Because you are reading an audio signal in this code, you will need to do some additional calculations to get useful information. An audio signal is changing fairly quickly, and the value returned by analogRead will depend on what point in the undulating signal you take a reading. If you are unfamiliar with using analogRead,
- An example waveform for an audio tone is shown in following figure. As time changes from left to right, the voltage goes up and down in a regular pattern. If you take readings at the three different times marked on it, you will get three different values.
- If you used this to make decisions, you might incorrectly conclude that the signal got louder in the middle.

6. Microphone- Detecting Sound

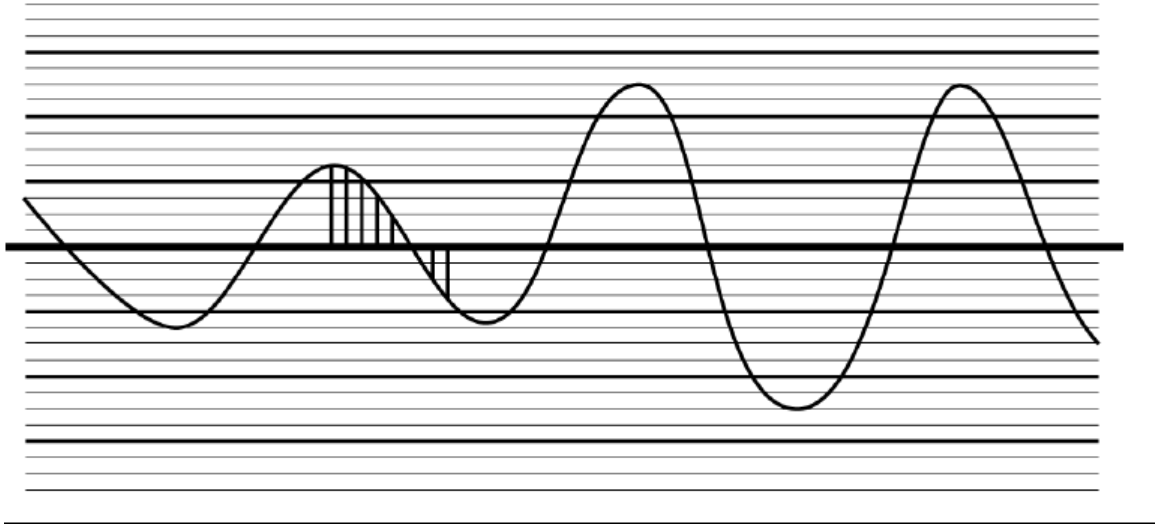
- An accurate measurement requires multiple readings taken close together. The peaks and troughs increase as the signal gets bigger. The difference between the bottom of a trough and the top of a peak is called the amplitude of the signal, and this increases as the signal gets louder.



Audio signal measured in three places

6. Microphone- Detecting Sound

- To measure the size of the peaks and troughs, you measure the difference between the midpoint voltage and the levels of the peaks and troughs. You can visualize this midpoint value as a line running midway between the highest peak and the lowest trough, as shown in following figure. The line represents the DC offset of the signal (it's the DC value when there are no peaks or troughs). If you subtract the DC offset value from your analogRead values, you get the correct reading for the signal amplitude.



Audio signal showing DC offset (signal midpoint)

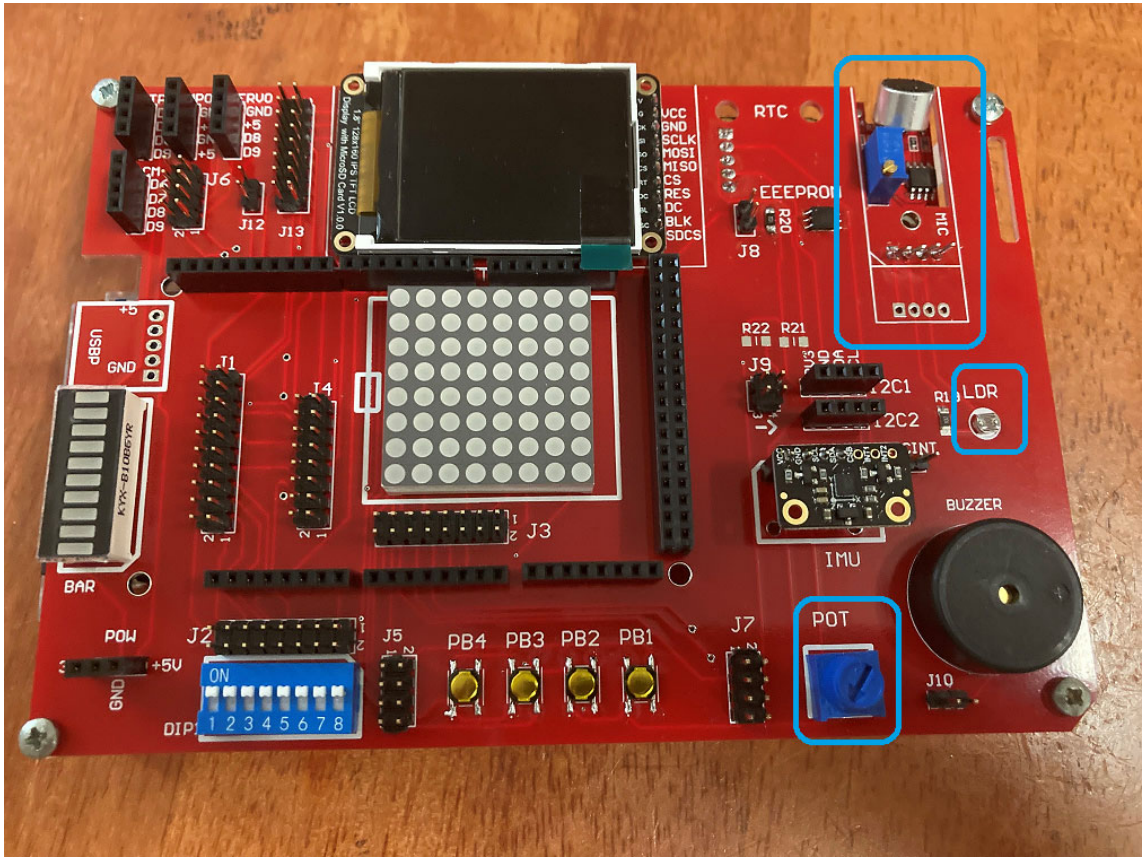
6. Microphone- Detecting Sound

- As the signal gets louder, the average size of these values will increase, but as some of them are negative (where the signal has dropped below the DC offset), they will cancel each other out, and the average will tend to be zero. To fix that, we square each value (multiply it by itself). This will make all the values positive, and it will increase the difference between small changes, which helps you evaluate changes as well. The average value will now go up and down as the signal amplitude does.
- To do the calculation, we need to know what value to use for the DC offset. To get a clean signal, the amplifier circuit for the microphone will have been designed to have a DC offset as close as possible to the middle of the possible range of voltage so that the signal can get as big as possible without distorting. The code assumes this and uses the value 512 (right in the middle of the analog input range of 0 to 1,023).

6. Microphone- Detecting Sound

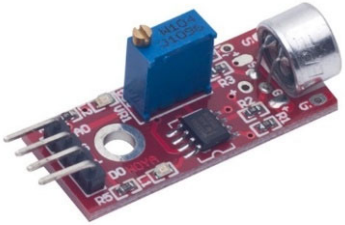
- The values of variables at the top of the sketch can be varied if the sketch does not trigger well for the level of sound you want.
- The numberOfSamples is set at 128—if it is set too small, the average may not adequately cover complete cycles of the waveform and you will get erratic readings. If the value is set too high, you will be averaging over too long a time, and a very short sound might be missed as it does not produce enough change once a large number of readings are averaged. It could also start to introduce a noticeable delay between a sound and the light going on. Constants used in calculations, such as numberOfSamples and averaged Over, are set to powers of 2 (128 and 16, respectively). Try to use values evenly divisible by two for these to give you the fastest performance (see Chapter 3 for more on math functions).

6. Potentiometer, LDR and MIC



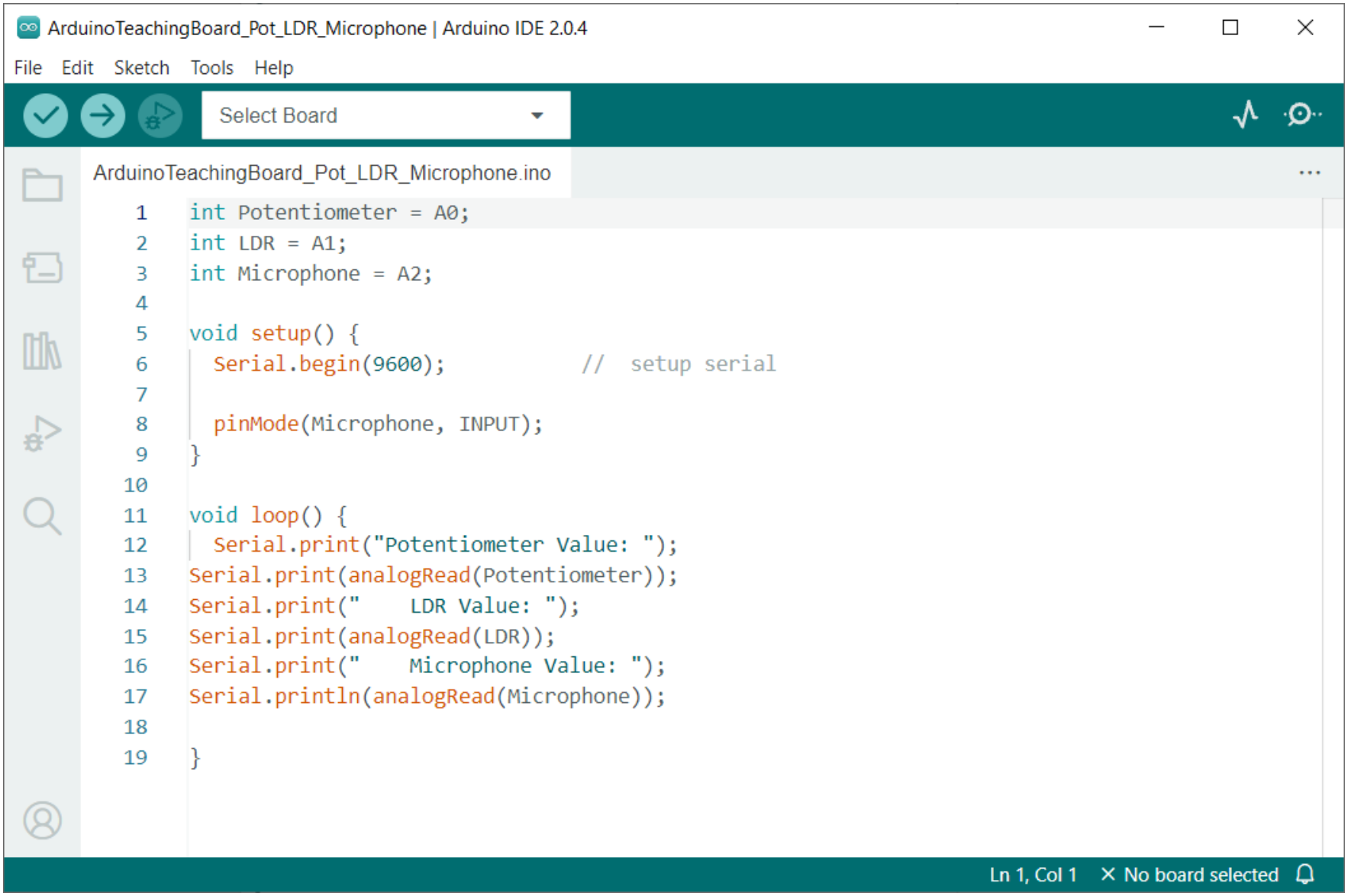
10K Ohm square trimming potentiometer

NSL-19M51
Light Dependent Resistor (LDR)



Audio Microphone Module

6. LDR and MIC


ArduinoTeachingBoard_Pot_LDR_Microphone | Arduino IDE 2.0.4
File Edit Sketch Tools Help
Select Board
ArduinoTeachingBoard_Pot_LDR_Microphone.ino
1 int Potentiometer = A0;
2 int LDR = A1;
3 int Microphone = A2;
4
5 void setup() {
6 Serial.begin(9600); // setup serial
7
8 pinMode(Microphone, INPUT);
9 }
10
11 void loop() {
12 Serial.print("Potentiometer Value: ");
13 Serial.print(analogRead(Potentiometer));
14 Serial.print(" LDR Value: ");
15 Serial.print(analogRead(LDR));
16 Serial.print(" Microphone Value: ");
17 Serial.println(analogRead(Microphone));
18
19 }
Ln 1, Col 1
No board selected

Commonwealth of Australia
Copyright Act 1968

Notice for paragraph 135ZXA (a) of the *Copyright Act 1968*

Warning

This material has been reproduced and communicated to you by or on behalf of Swinburne University of Technology under Part VB of the *Copyright Act 1968* (the ***Act***).

The material in this communication may be subject to copyright under the Act. Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.