EPISODE 2

I/O Operations

Arduino pins are by default configured as inputs, so they do not need to be explicitly declared as inputs with **pinMode()** when you are using them as inputs. Pins configured this way are said to be in a high-impedance state. Input pins make extremely small demands on the circuit that they are sampling, equivalent to a series resistor of 100 megohm in front of the pin. This means that it takes very little current to switch the input pin from one state to another. This makes the pins useful for such tasks as implementing a capacitive touch sensor or reading an LED as a photodiode.

Pins configured as pinMode(pin, INPUT) with nothing connected to them, or with wires connected to them that are not connected to other circuits, report seemingly random changes in pin state, picking up electrical noise from the environment, or capacitively, coupling the state of a nearby pin.

Pins configured as OUTPUT with pinMode() are said to be in a low-impedance state. This means that they can provide a substantial amount of current to other circuits. Atmega pins can source (provide positive current) or sink (provide negative current) up to 40 mA (milliamps) of current to other devices/circuits. This is enough current to brightly light up an LED (do not forget the series resistor), or run many sensors but not enough current to run relays, solenoids, or motors. Attempting to run high current devices from the output pins, can damage or destroy the output transistors in the pin, or damage the entire Atmega chip. Often, this results in a "dead" pin in the microcontroller but the remaining chips still function adequately. For this reason, it is a good idea to connect the OUTPUT pins to other devices through 470 or 1k resistors, unless maximum current drawn from the pins is required for a particular application.

```
pinMode(pin, mode);

//pin: pin number of your wish to connect

//mode: INPUT, OUTPUT, INPUT_PULLUP

// analog pins unlike digital pins doesn’t require
declaration as input or output
```

Pull-up Resistor

Using Built-in Pull-up Resistor with Pins Configured as Input often useful to steer an input pin to a known state if no input is present. This can be done by adding a pull-up resistor (to +5V), or a pull-down resistor (resistor to ground) on the input. A 10K resistor is a good value for a pull-up or pull-down resistor. There are 20,000 pull-up resistors built into the Atmega chip that can be accessed from software. These built-in pull-up resistors are accessed by setting the **pinMode()** as INPUT_PULLUP. This effectively inverts the behaviour of the INPUT mode, where HIGH means the sensor is OFF and LOW means the sensor is ON. The value of this pull-up depends on the microcontroller used. On most AVR-based boards, the value is guaranteed to be between 20k and 50k. On the Arduino Due, it is between 50k and 150k. For the exact value, consult the data sheet of the microcontroller on your board.

```
pinMode(pin, INPUT_PULLUP);
```

I/O Functions

```
digitalWrite(pin, value);
digitalRead(pin); // Returns a bool value 1 or 0

analogWrite(pin, value); // value proportional to duty cycle
analogRead(pin); // Returns value between 0-1023 or 0-255
                 //depending on value read from ADC or ADCH

analogReference(type);
//type - INTERNAL, EXTERNAL, DEFAULT,
//INTERNAL1V1, INTERNAL2V56

data_type variable = pulseIn(pin, value); // value: HIGH or
LOW

tone(pin, frequency, duration); // freq in Hz, duration in ms
```

Serial Communication

Serial communications provide an easy and flexible way for your Arduino board to interact with your computer and other devices. This chapter explains how to send and receive information using this capability. You can also send data from the Serial Monitor to Arduino by entering text in the text box to the left of the Send button. Baud rate (the speed at which data is transmitted, measured in bits per second) is selected using the drop-down box on the bottom right. You can use the drop down labeled “No line ending” to automatically send a carriage return or a combination of a carriage return and a line at the end of each message sent when clicking the Send button, by changing “No line ending” to your desired option.

Your Arduino sketch can use the serial port to indirectly access (usually via a proxy program written in a language like Processing)
all the resources (memory, screen, key- board, mouse,network connectivity, etc.) that your computer has. Your computer can also use the serial link to interact with sensors or other devices connected to Arduino.

Serial Protocols

The hardware or software serial libraries handle sending and receiving information. This information often consists of groups of variables that need to be sent together. For the information to be interpreted correctly, the receiving side needs to recognise where each message begins and ends. Meaningful serial communication, or any kind of machine-to-machine communication, can only be achieved if the sending and receiving sides fully agree how information is organised in the message. The formal organisation of information in a message and the range of appropriate responses to requests is called a *communications protocol*.

```
Serial.begin(BAUD_RATE); // BAUD_RATE: eg. 9600, 115200 bps

// flush the receive buffer
Serial.flush(); // or
while(Serial.read() >= 0);

Serial.write(); // usually followed by flush function

Serial.read(); // returns char removed from buffer

Serial.peek(); // peeks at the next char in buffer but
               // doesn't remove it from buffer

Serial.print(value, ENCODING); // doesn't end with newline
                               //char, ENCODING: BYTE, BIN,
                               //HEX, OCT, DEC

Serial.println(); // with end line char
```

```
serialEvent(); // function is called in void loop
```

PROGRAM_1[POT]

```
const int potPin = 0; // select the input pin for the
potentiometer
const int ledPin = 13; // select the pin for the LED

void setup()
{
  pinMode(ledPin, OUTPUT); // declare the ledPin as an OUTPUT
  Serial.begin(9600);
}
void loop() {
  int val; // The value coming from the sensor
  int percent; // The mapped value

  val = analogRead(potPin); // read the voltage on the pot
                            //(val ranges from 0 to 1023)
  percent = map(val, 0, 1023, 0, 100); // percent will range from
                                       //0 to 100.

  digitalWrite(ledPin, HIGH); // turn the ledPin on
  delay(percent); // On time given by percent value
  digitalWrite(ledPin, LOW); // turn the ledPin off
  delay(100 - percent); // Off time is 100 minus On time

  Serial.println(percent); // show the % of pot rotation on Serial
                           //Monitor
}
```

PROGRAM_1.1[Serial COM]

```
char chrValue = 65; // these are the starting values to print
int intValue = 65;
float floatValue = 65.0;

void setup()
{
  Serial.begin(9600);
}
void loop()
{
  Serial.println("chrValue: ");
  Serial.println(chrValue);

  Serial.println(chrValue, DEC);
  Serial.println("intValue: ");
  Serial.println(intValue);

  Serial.println(intValue, DEC);
  Serial.println(intValue, HEX);
  Serial.println(intValue, OCT);
  Serial.println(intValue, BIN);
  Serial.println("floatValue: ");
  Serial.println(floatValue);

  delay(1000); // delay a second between numbers
  chrValue++; // to the next value
  intValue++;
}
```

PROGRAM_1.2[Embed-C]

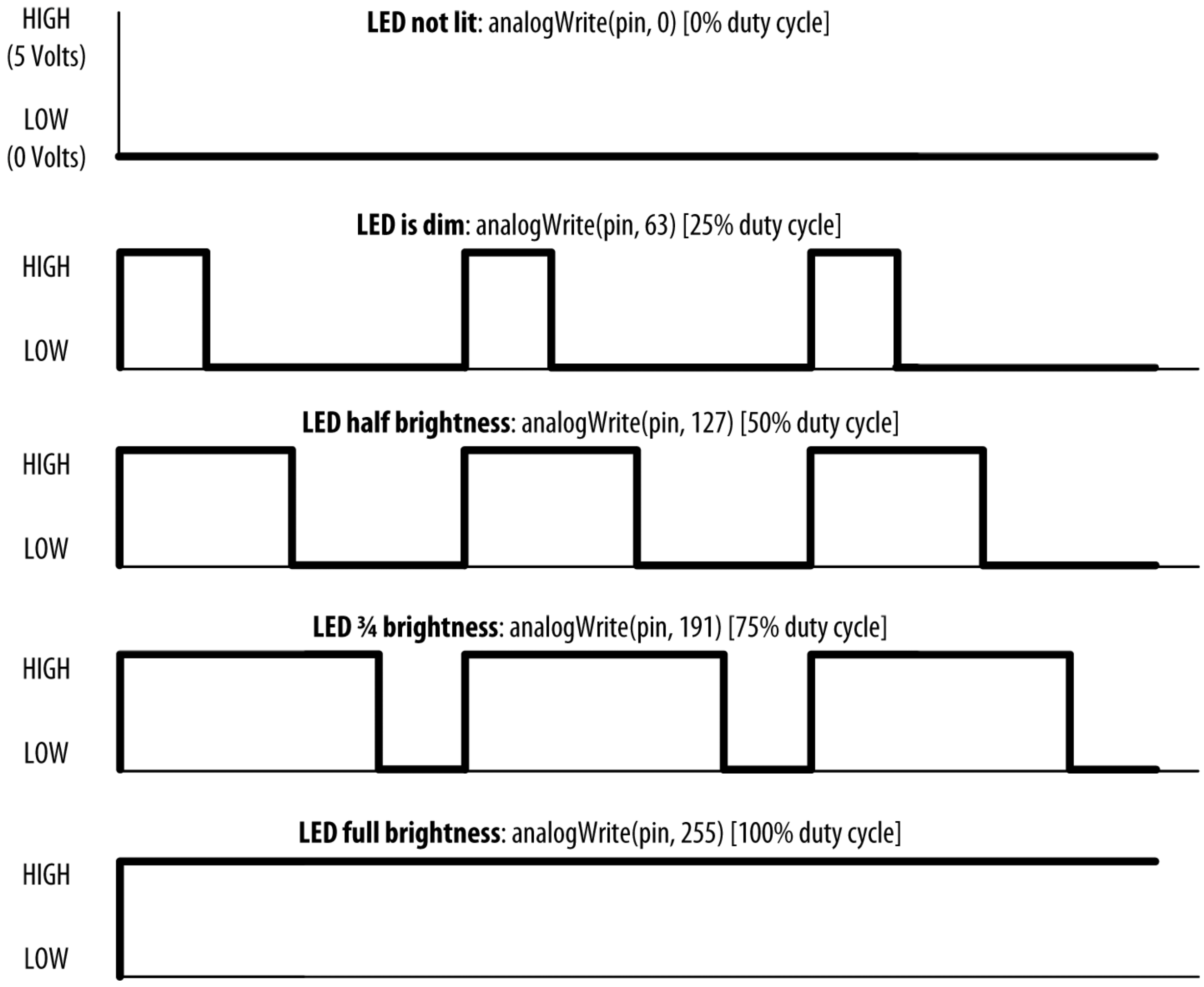
```
unsigned int adc_v;

int main() {
  Serial.begin(115200);
  DDRC &= 0x00;
  PORTC |= 0x11;

  ADMUX |= 1 << REFS0; //avcc
  ADMUX |= (1 << MUX0) | (1 << MUX2);
  ADCSRA |= (1 << ADPS0) | (1 << ADPS1) | (1 << ADPS2);
  ADCSRA |= 1 << ADEN; //adc enable
  ADCSRA |= 1 << ADIE;
  ADCSRA |= 1 << ADSC; //call adc

  sei();
  while (1);
  return 0;
}

ISR(ADC_vect) {
  adc_v = ADC;
  Serial.println(adc_v);
  _delay_ms(100);
  ADCSRA |= 1 << ADSC;
}
```

PWM

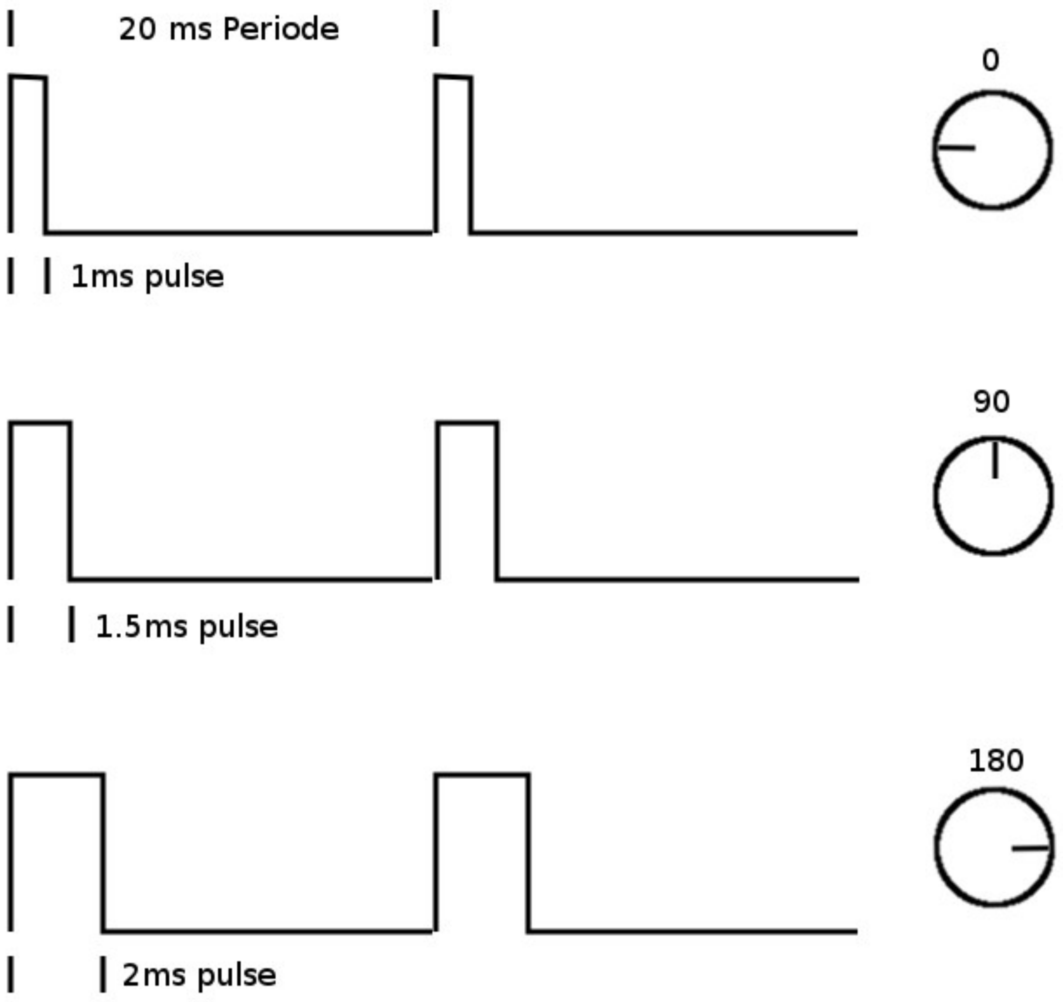
The analogWrite function is not truly analog, although it can behave like analog, as you will see. analogWrite uses a technique called Pulse Width Modulation (PWM) that emulates an analog signal using digital pulses.

PWM works by varying the proportion of the pulses' on time to off time, Low-level output is emulated by producing pulses that are on for only a short period of time. Higher level output is emulated with pulses that are on more than they are off. When the pulses are repeated quickly enough (almost 500 times per second on Arduino), the pulsing cannot be detected by human senses, and the output from things such as LEDs looks like it is being smoothly varied as the pulse rate is changed.

Arduino has a limited number of pins that can be used for analog output. On a standard board, you can use pins 3, 5, 6, 9, 10, and 11. On the Arduino Mega board, you can use pins 2 through 13 for analog output. Many of the recipes that follow use pins that can be used for both digital and analog to minimise rewiring if you want to try out different recipes. If you want to select different pins for analog output, remember to choose one of the supported analogWrite pins (other pins will not give any output).


HIGH (5 Volts)
LOW (0 Volts)
LED not lit: analogWrite(pin, 0) [0% duty cycle]
LED is dim: analogWrite(pin, 63) [25% duty cycle]
HIGH
LOW
LED half brightness: analogWrite(pin, 127) [50% duty cycle]
HIGH
LOW
LED ¾ brightness: analogWrite(pin, 191) [75% duty cycle]
HIGH
LOW
LED full brightness: analogWrite(pin, 255) [100% duty cycle]
HIGH
LOW

SERVO



PROGRAM_2 [Micro_servo_basic]

```
#include<Servo.h>
Servo myservo;
void setup(){
  myservo.attach(9);
}
void loop(){
  myservo.write(90);
  delay(500);
  myservo.write(0);
  delay(500);
}
```

PROGRAM_2.1[Servo]

```
#include<Servo.h>
Servo myservo;
void setup(){
  myservo.attach(9);
}

void loop(){
  for(int i=0;i<=180;i+=30){
    myservo.write(i);
    delay(300);
  }


  for(int i=180;i>=0;i-=30){
    myservo.write(i);
    delay(300);
  }
}
```

PROGRAM_2.2 [Light Meter]

```
#include <Servo.h>

Servo myservo;

const int apin = 0;
int val;

void setup() {
  myservo.attach(9);
}

void loop() {
  val = analogRead(apin);
  val = map(val, 0, 1023, 0, 180);
  myservo.write(val);
  delay(10);
}
```