EduCake Analog I/O Intro

1. Introduction to Analog I/O

In previous chapter, we introduced the 86Duino EduCake, talked about EduCake's I/O features and specification, the development IDE and multiple examples showing how to use EduCake's digital I/O. In this chapter, we will talk about EduCake's analog I/O.

• Analog versus Digital

Prior to introducing analog I/O, let's take a look at the difference between digital I/O and analog I/O. In general, a digital I/O pin has two possible value, represented by two different electrical status, high-voltage condition and low-voltage condition. When a digital I/O pin's measured voltage is above a certain level, it's recognized as high and when the digital I/O pin's measured voltage is below a certain level, it's recognized as low. Within the 86Duino development environment, the digitalWrite () function is used to set a digital I/O pin to high or low, and the digitalRead () function is used to retrieve a digital I/O pin's High or Low status, which can be used to detect whether a switch, to monitor a point of entry, is in open or closed condition. Digital I/O is also suitable to turn things on and off, such as electric appliances, via a relay.

Different from digital I/O with 2 possible value, analog I/O is an electrical signal within a range of values. Analog input is commonly used as input for different type of sensors, such as temperature, humidity, pressure and light intensity measurement. However, computing device is not able to read an analog input directly and requires an analog to digital converter (ADC) to convert analog signal to digital value, represent by binary numbers, as shown in a sample ADC circuit in Fig-1 below:

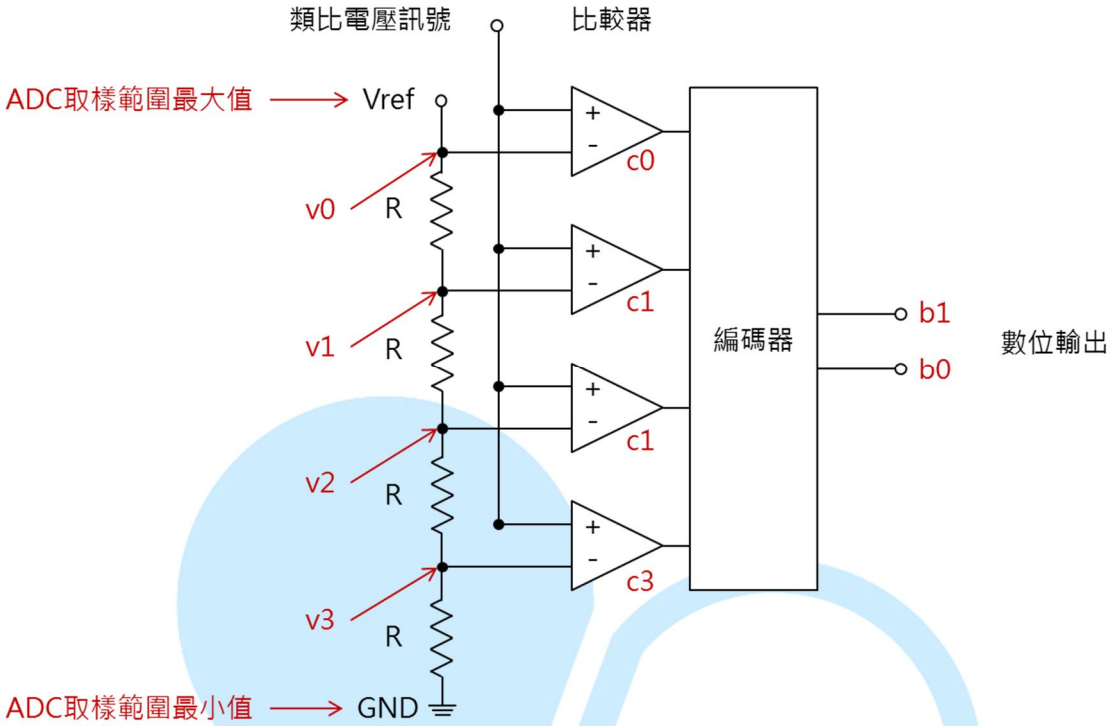


Fig-1. 「Flash ADC」

The circuit in Fig-1 above is an easy to understand Flash ADC example. There are 4 comparators, along with reference voltage (Vref), to compare with the analog input signal. The analog input is directed to each of the comparator. As shown in Fig-01, 4 different voltages, v0, v1, v2 and v3 are directed to each of the comparators. When the analog input voltage is higher than v0, c0 will output 1. When the analog input voltage is higher than v1, c1 will output 1. When the analog input voltage is higher than v2, c2 will output 1. When the analog input voltage is higher than v3, c3 will output 1. The Flash encoder is responsible for translating the output value from c0, c1, c2 and c3 into the binary output, b0 and b1, a 2-bit binary value to represent the 4 different possible values. Additional bits are needed to represent more possible values, such as a 4-bit binary output is needed to represent 16 different possible values.

With 5V as Vref, the voltage value for v0~v3 are 5V, 3.75V, 2.5V and 1.25V. When an analog input of 1.5V is present, the c0~c3 comparators would output 0/0/0/1, which translate to binary value 01. When an analog input of 3.9V is present, the c0~c3 comparators would output 0/1/1/1, which translate to binary value 11. This is one of the common method, using multiple comparators to translate analog input to a binary value. Analog to digital conversion is a broad and challenging subject. The example provided here is a simple one to help you understand the basic. You need to read more about ADC to get deeper understanding.

• ADC Specification and Conversion Method

In general, specification for an ADC device typically specify the input range and output resolution. As presented in the sample in the previous section, when the input is higher or lower than the ADC specified input range, it will be represented by the highest and lowest output values. Additional output bits will yield higher number of possible values which affect the resolution and accuracy of the translated digital value.

Example 1: An ADC with 0~5V and 10 bits resolution will output the value "000000000" when the input is at 0V and will output the value "1111111111" when the input is at 5V. At 10 bits resolution, within the 0 to 5V range, there are 1024 possible values, which means this ADC device is capable to measure voltage value in 0.004883 increment, within the 0 to 5V range (5 / 1024). An ADC translated output equivalent to the 1000th value is equal to 4.883V (1000 * (5 / 1024)).

Example 2: An ADC with 1~7V and 12 bits resolution has 4096 possible values, at 0.00146V resolution. An ADC translated output equivalent to the 1000th value is equal to 2.465V (1000 * ((7-1)/4096)).

When using an ADC, it's important to understand it's specification and resolution. The 86Eduino EduCake's Analog input range is 0~3.3V with 10 bits resolution.

• Things you should know about Sensor

In the previous section, we talk about the analog signals from sensor and using ADC to translate analog signals to binary values that can be read by computing device. It's important to know that many sensors' output may not meet the ADC's input specification and requires additional circuitry to condition the sensor's output prior to the ADC, as discuss in the following examples.

Example 1: Resistive sensor device, such as pressure sensor, lighting sensor, temperature sensor and etc., function like a variable resistor, where the sensor device's resistive value change relative to the environment the device is sensing. When use as shown in circuits in Fig-2, the Vcc for these resistance voltage-divider sensing method is very close to the highest value within the range of ADC's supported input, such as the 3.3V input for the EduCake. As shown in Fig-2, the input voltage for ADC is equal to (Vcc * R1 / (R1 + R2)). In Fig-2b, the combined R1 + R2 value, for the variable resistor, is static. In Fig-2a, you can use different resistant value for R2, recommend to use

approximately 1K Ohm range to control current flow along with a suitable R1 value to condition the analog input signal to be within the ADC's supported input voltage range.

Fig-2(a) Fig-2(b)

Example 2: It's common for sensor such as Pyrometer, to measure thermal radiation, has a 4~20mA output. This type of sensor output a variable current relative to the sensor value. To use this type of sensor, connect a resistor to the sensor's output, as shown in Fig-3. To convert 4~20mA signal to 1~5V range, a 250 Ohm resistor is needed (V = I * R). When working with 4~20mA sensor, pay close attention to the current flow direction.

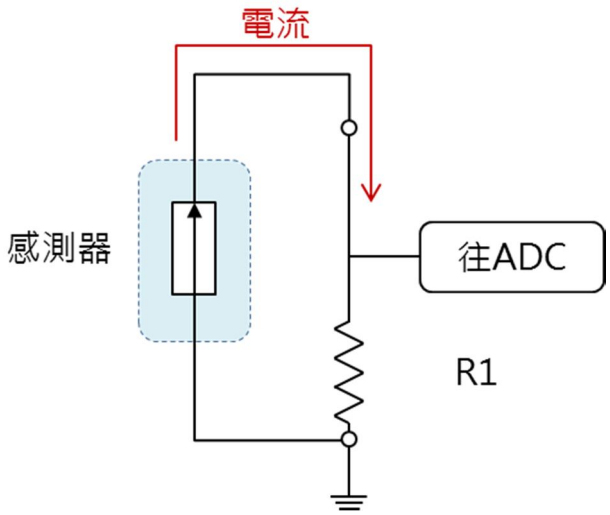


Fig-3

• Analog Output

In the previous section, we talked about ADC, analog input and converting analog input to digital value. Working with analog output is just the opposite, which involves using a digital to analog converter (DAC) to convert digital output to analog value. Not all devices require a pure analog signal to function, such as light bulb and DC motor. A signal with proper voltage and proportion of current, a light bulb's brightness and the speed for a DC motor can be accurately controlled. The EduCake provides a number of PWM (Pulse Width Modulation) output signals to use as analog outputs. In addition to serving as analog output, PWM has other useful function which will be covered in the later chapters.

As shown in Fig-4, from top to bottom, there are 4 graphs showing different duty cycle for PWM signal, at 25%, 50%, 75% and 100% level. Since human's vision retain short term image, at high frequency, PWM voltage apply to the light bulb reflect the proportioned intensity. The PWM signal on the EduCake is at 1000 Hz. Instead of using PWM, it's possible to use the digitalWrite() along with the delay() and delayMicroseconds() function to create similar function. However, when the frequency is slow, the lighting will show some flickering.

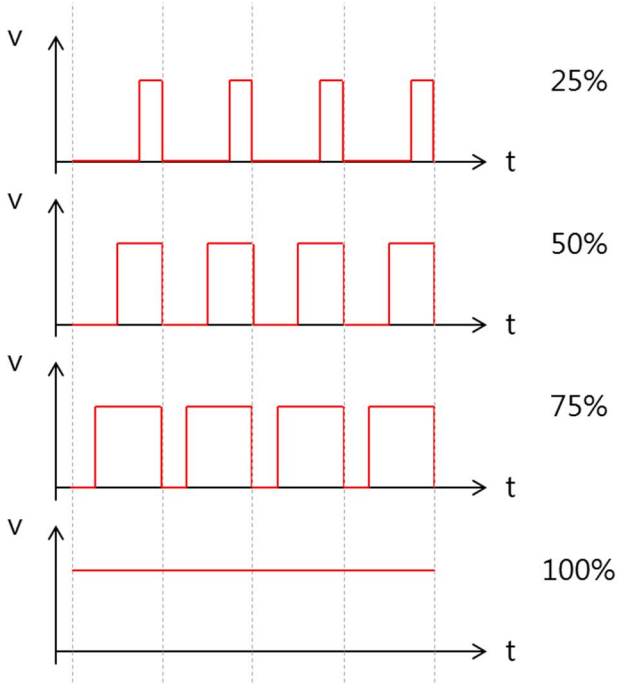
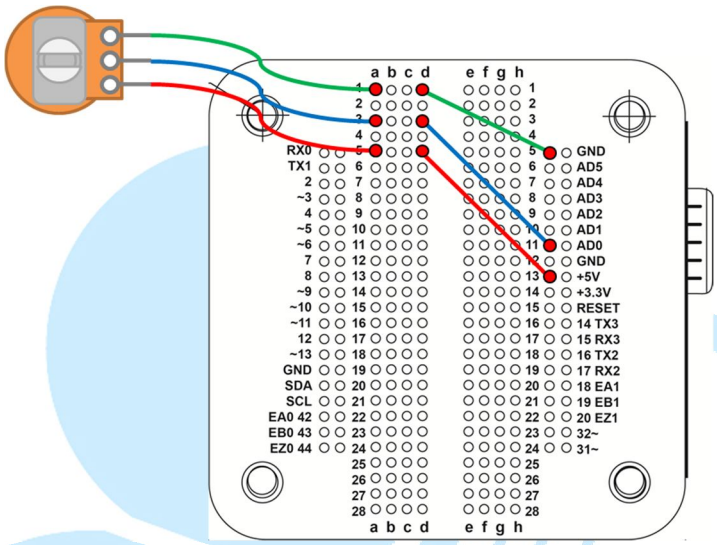


Fig-4

With the introduction out of the way, we will work through a number of hands on exercises in the following sections.

2. First Exercise – analogRead()

For this exercise, we will use EduCake to perform simple analog input, using the analogRead() function. A variable resistor (10K Ohm recommended) and a few wires are needed to create the circuit for the exercise, as shown in the following figure.



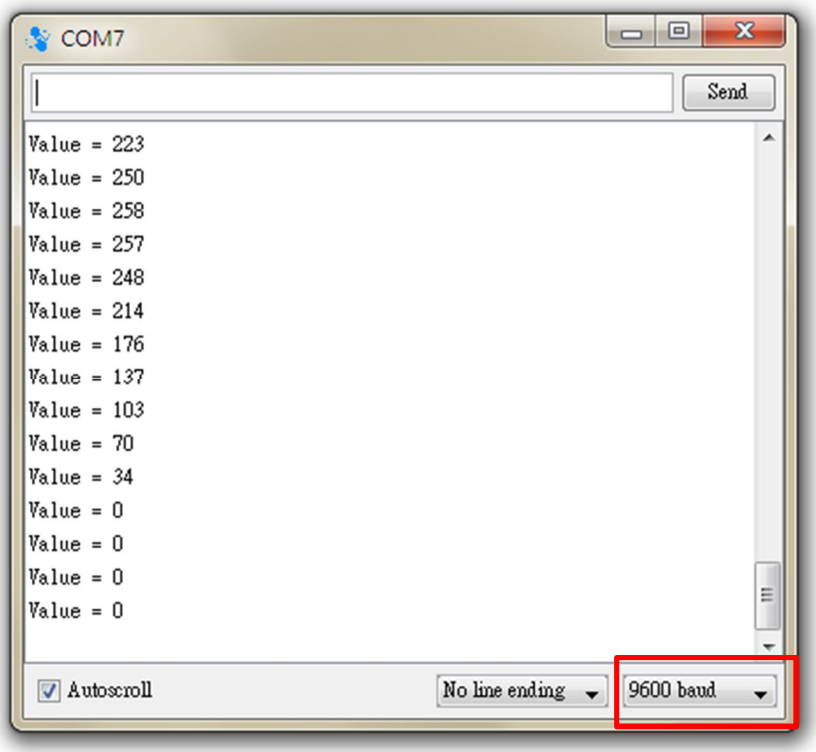
The circuit for this exercise is similar to the resistive voltage divider we talked about in the earlier section, as shown in Fig-2b. Pay attention to the variable resistor's 3 different pins. The middle pin is used to change the resistance value. Connect the other 2 pins to the EduCake's 3.3V and GND. Although these 2 pins are not polarity sensitive, it's coordinated with the variable resistor's rotation direction that is in proportion with the changing resistance value, from low to high or high to low. In this example, we will use an ADC input to read the value and output the value to the development workstation for viewing, via the USB connectivity to the development workstation.

Launch the 86Duino development IDE and enter the following codes:

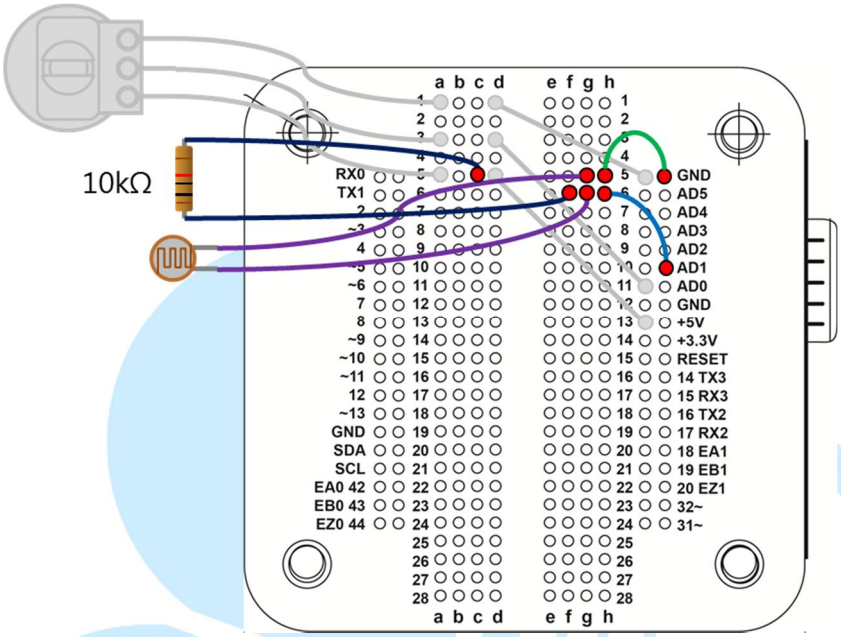
```
const int analogInPin = A0;// Declare and assign analog input pin A0 (AD0)

int sensorValue = 0;// Declare variable to hold sensor reading

void setup() {
  Serial.begin(9600);// Configure and setup serial port
}

void loop() {
  sensorValue = analogRead(analogInPin);// Read analog input

  Serial.print("Value = " );// Output sensor value via serial port
  Serial.println(sensorValue, DEC);// Output sensor value via serial port

  delay(100);//
}
```

When the above code is executed, the setup () function is call first to setup the baudrate, which is the communication speed for the serial port. Then, it execute the program loop(), which run continuously, calling the analogRead() function to read data from the analog input and calling the Serial.print() and Serial.println() functions to output the analog data through the serial port, which can be view via Serial Monitor. In addition to output the assigned value to the serial port, the Serial.println() function add a line feed to advance to the next line. The DEC variable within the "`serial.println(sensorValue, DEC)`" line of code set the output to decimal value. You can change this variable to BIN or HEX to set the output value to binary or HEX. The delay(100) function simply delay the function loop by 100 ms and continue to execute the code in the function loop again, which can change the value to adjust the delay time.

After deploying the above code to the 86Duino EduCake, select Tools→Serial Monitor (or use the Ctrl+Shift+M shortcut) to launch the Serial Monitor and view the analog data read from the ADC, as shown in the following figure. Next, you can turn the mechanical wheel on the variable resistor to change the resistance value between 0 ~ 1023. From the Serial Monitor, you should see the different analog value as the resistance is changed. If the Serial Monitor is not showing data and information as expected, check the Serial Monitor screen's lower right, to make sure it's configured to the correct baudrate.



Next, we will continue the exercise with a different sensor, based on the circuit as shown in Figure-2a. Keeping the existing circuit for the later exercises, we will leave the existing circuit in place and add additional components, as shown in the following figure:



A photoresistor sensor, a common components within the DIY market, is used for the next exercise. We will use an indoor lighting photoresistor sensor with 10 k ohm variable resistance with a fixed 10 k ohm resistor. One of the photoresistor's output is connected to GND, the other output is connected to AD1. One of the output from the fixed 10 k ohm resistor is connected to 5V, the other output is also connected to AD1, as shown in the figure above. When the photoresistor is illuminated, the resistance value decreases, when not illuminated, the resistance value increases (which may be as high as mega ohm). Based on the above circuit, readings from the ADC is expected to reduce as the lighting brightness increases, and readings from the ADC is expected to increase when the photoresistor is covered from the light source.

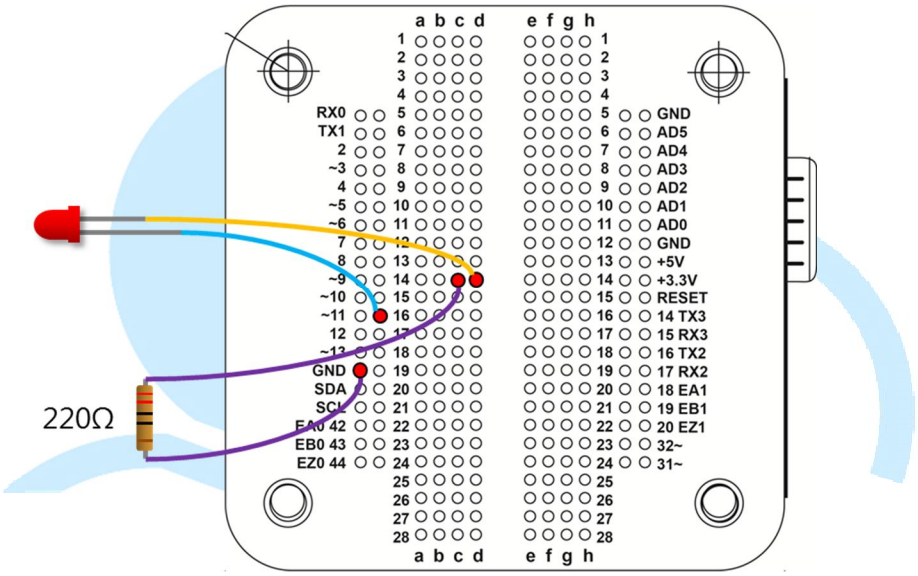
Using the existing code from the earlier exercise, we can simply modify the following line of code:

```
const int analogInPin = A1;// Declare and assign analog input pin A1 (AD1)
```

Basically, the above code changes the analog input to AD1. You can view the result using the same procedure as in the previous exercise. As you can see, a simple circuit with minor change can be adopted to deliver different results.

3. Second Exercise – analogWrite()

In this next exercise, we will be working with a flickering LED circuit to explore the analog output function. An LED, in either yellow, red or green would work, along with a 220 ohm resistor to protect the LED by limiting the current flow, as shown in the circuit in the following figure. The existing circuit from the previous exercises are kept in place. The following circuit is added on to the existing circuit.



The LED's positive pin (with longer lead) is connected to pin 11 on the EduCake's breadboard. All of the pins marked with the "~" symbol are capable to generate PWM output.

From the 86Duino development IDE, enter the following codes:

```
const int analogOutPin = 11;// Declare and assign analog output pin 11
const float Freq = 0.5;// LED flashing frequency
unsigned long time;// time variable
byte LED_light = 0;// variable to control LED brightness

void setup() {
}

void loop() {
  time = millis();

  // Calculate LED brightness
  LED_light = byte(128 + 127 * cos((float)time * 0.001 * 2 * PI * Freq));

  //Serial.print("LED_light = " );// output to serial port
  //Serial.println(LED_light, DEC);// output LED brightness in decimal
  analogWrite(analogOutPin, LED_light);//output PWM to LED signal in waveform

  delay(50);// time delay
}
```

In the above code, the first 4 lines of codes declare and initialize variables for the program, such as analog output pin, LED flashing frequency, time and LED brightness. For each iteration in the main program loop, value for the time variable is set using the millis() function to retrieve the time in milli-seconds the application has been running. Then, using the Cosine function (The

Sine function can be used as well) in the formula below, to calculate the value used to control the LED brightness.

LED brightness = 128 + 127 * cos(2 * π * freq * time)

The output range for PWM using the analogWrite() function is 0~255, where 255 represent 100% duty cycle, 128 represent 50% duty cycle and 0 represent 0%. The output value from the cos() function in the above formula is between -1 to +1. Multiplying the result from the cos() function by 127 and adding 128 will yield a value between 1 ~ 255. The 2πft is used to adjust the cos() function's frequency. The value for f is in Hz. Since value for the time variable from the millis() function is in millisecond, multiply by 0.001 will convert the value into second. In the last line of code, the delay() function add time delay before repeating the main program loop again.

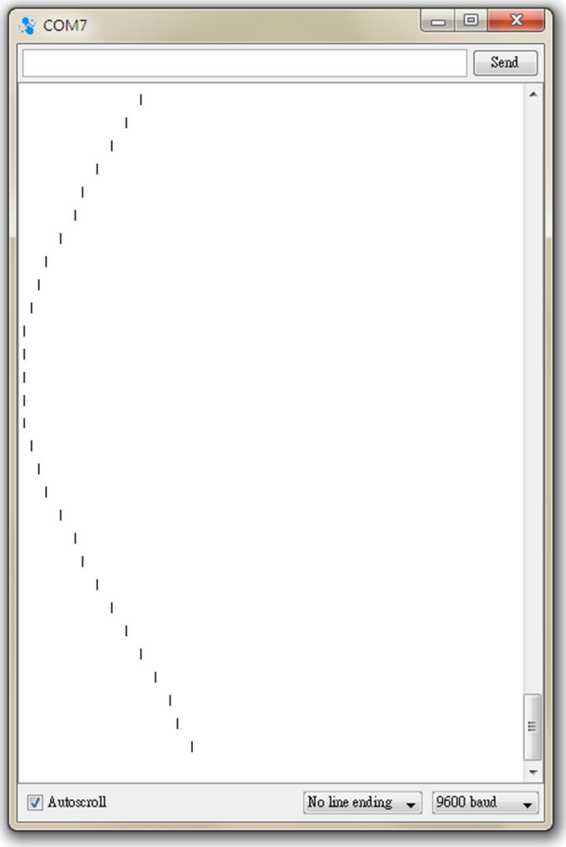
This example set the LED flashing frequency to 0.5 Hz, when executing the code, you can see the LED flickering from bright to dark every 2 seconds. You can change the value in the delay() function, without changing the LED flickering frequency from bright to dark. However, you will notice the change is not smooth. This example uses the 50ms delay time, along with 20Hz as the frequency for the LED brightness calculation formula, the flickering effects is much better.
In addition, we will demonstrate another usage for the Serial Monitor, by changing the following lines of code:

```
//Serial.print("LED_light = " );// output to serial port
//Serial.println(LED_light, DEC);// output LED brightness in decimal
```

To the following:

```
for(int i=0;i<LED_light/10;i++){
  Serial.print(" " );// output to serial port
}
Serial.println("|");//output to serial port with line feed
```

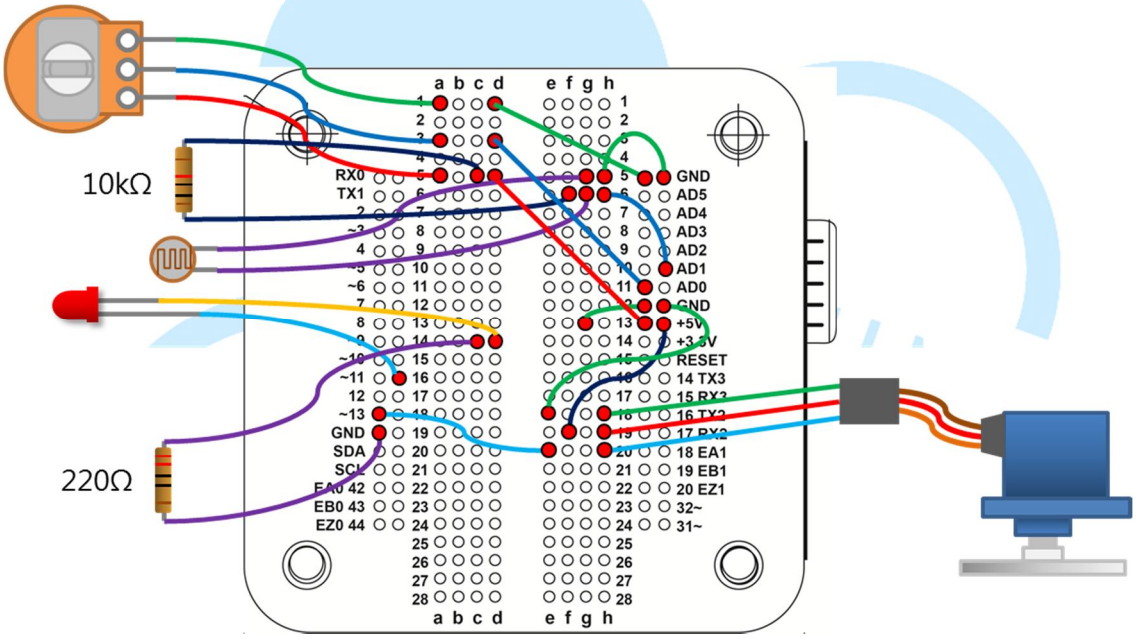
The above modified code uses the Serial.print() function to output blank space according to the LED brightness value, follow by the Serial.println() function to output a "|" symbol and line feed to advance to the next line, which result in a waveform output relative to the changes to the LED brightness, as shown in the following figure:



The above waveform is synchronize with the LED's flickering intensity. You can change the formula to see different variation.

4. Last Exercise for this chapter

In this last exercise, we will combine the knowledge from the previous exercises in this chapter, with additional components, to create a more interesting example. A servo motor is needed for this exercise. For the example, we use a Tower Pro SG90 servo, a small servo used in different type of remote control, also refer to as RC servo. In general, larger size RC servo requires higher voltage and current to operate. For this exercise, we selected the SG90 servo to work with the EduCake's 5V power source, without the need to add external power to drive the servo. With the circuits from the previous exercises intact, connect the servo to EduCake as shown in the following figure:



There are three wires attached to the SG90 servo, the brown wire is connected to GND, red wire is connected to 5V power source and orange wire is connected to the control signal, in this case the PWM signal. For this exercise, the orange wire is connected to pin 13 on the EduCake's breadboard.

Enter the following code for the exercise:

```
#include <Servo.h>

const int Vr_analogInPin = A0;// Declare & assign AD0 to variable resistor
const int Ls_analogInPin = A1;// Declare & assign AD1 to photoresistor
const int LED_analogOutPin = 11;// Declare & assign pin 11 to LED
const int Servo_Pin = 13;// Declare & assign pin 13 to servo

int Light_threshold = 0;

Servo Servo_1;

void setup() {
  Serial.begin(9600);
  Servo_1.attach(Servo_Pin);//, 500, 2400
  Servo_1.write(90);// Move servo to center position
```

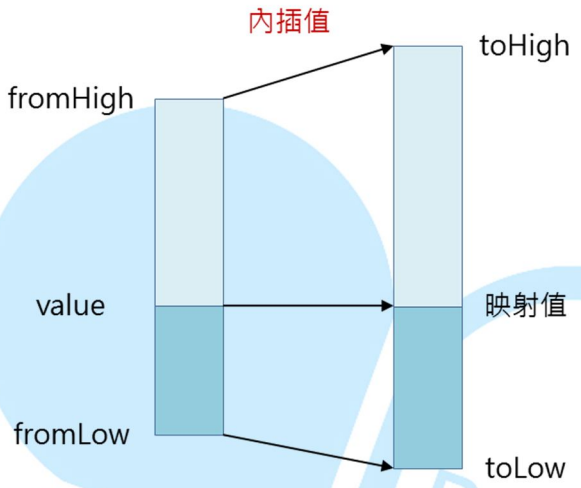
```
  // Retrieve photoresistor ADC value
  // Use the initial value as the lower limit
  Light_threshold = analogRead(Ls_analogInPin);

}

void loop() {
  // Retrieve ADC value from variable resistor
  int Vr_rsensorValue = analogRead(Vr_analogInPin);//

  //  Retrieve ADC value from photoresistor
  int Ls_rsensorValue = analogRead(Ls_analogInPin);//

  //  Set limit for reading from light sensor
  Ls_rsensorValue = constrain(Ls_rsensorValue, Light_threshold, 1023);//

  //  Set photoresister ADC value as LED light intensity
  byte LED_light = map(Ls_rsensorValue, 512, 1023, 0, 255);

  //  PWM Output to LED
  analogWrite(LED_analogOutPin, LED_light);

  //  Remap value from variable resistor ADC as servo position
  int Servo_1_angle = map(Vr_rsensorValue, 0, 1023, 0, 180);

  //  Output PWM signal to the servo
  Servo_1.write(Servo_1_angle);

  //  The following code, when uncommented, output program activities
  //  to the serial port.
  /*
  Serial.print("Vr Sensor value = " );
  Serial.print(Vr_rsensorValue, DEC);
  Serial.print(", Servo angle = " );
  Serial.print(Servo_1_angle, DEC);
  Serial.print(". Light Sensor value = " );
  Serial.print(Ls_rsensorValue, DEC);
  Serial.print(", LED light = " );
  Serial.println(LED_light, DEC);
*/

  delay(20);
}
```

Although the SG90 servo is controlled by PWM signal, the duty cycle is different from the analogWrite() function. In the above codes, the 1st line of code, “#include <Servo.h>”, brings in additional resources from the Servo.h header file, which provide additional servo related function. This example uses ADC value from the photoresistor to control LED brightness and uses ADC value from the variable resistor to control the RC servo, synchronizing the servo movement with the variable resistor rotation.

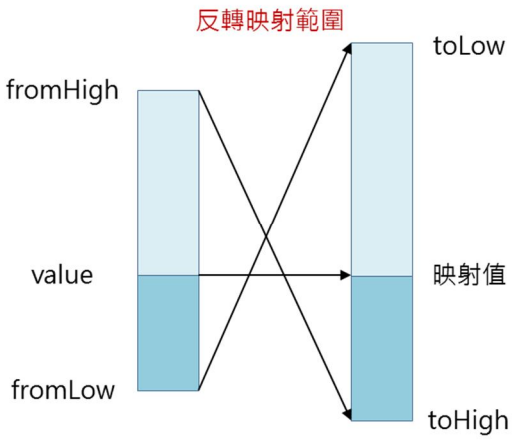
Here are some programming practices you need to pay attention to. Since the Servo.h is written with an object-oriented syntax, you need to declare a Servo object, such as Servo_1 and use the Servo_1.attach(Servo_Pin) function to assign an I/O pin (Servo_Pin) to control the servo. For each additional servo, you need to declare additional Servo object and add addition Servo.attach() function. Using Servo_1.Write() function, you can move the servo, associated with the Servo_1 object, to a particular position (angle). Typically, the center position is 90 degree and the other two sides are 0 and 180 degree. Search online to find relevant tutorial to

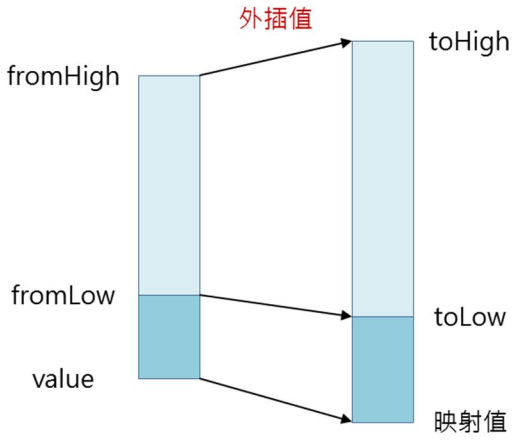
learn more about servo. For this exercise, we use the Servo.Write() function to control servo movement.

In the loop() function, the main program loop, ADC value from the photoresistor is read. Then, the map() function map the value, "`map(value, fromHigh, fromLow, toHigh, toLow)`", and return proportionally mapped value, as shown in the following figure.



In the map() function, the "value" variable represent the input, the "fromHigh", "fromLow", "toHigh" and "toLow" are the relative upper and lower limits ranges. For example, to map an ADC value of 127, reading from a device with 0~1023 upper and lower limits, to the 0~255 range, using the "y = map(127, 0, 1023, 0, 255)" function, the value for y is 31. The map() is not limited to positive value or the upper limit must be of higher value. You can use negative value with different variation of upper and lower limits ranges, as shown in the figure after this paragraph. The map() function uses integer arithmetic. When fractional number is used, it will be converted to integer before calculation. Going back to the LED flickering example, you can think about how to rewrite the code using the map() function.





Within the main program loop, loop(), there are two instances of the map() function, to map ADC value to the LED's brightness within the 0~255 range, and the servo's position within the 0~180 range. The map() function is useful to serve different purposes. In the sample program, to set the value for "Light_threshold", which is used as the initial value for the photoresistor when the application is launched. To set the LED to off condition when the application is launched and turn the LED to maximum brightness when the surrounding is in total darkness, the "Light_threshold" value is used as the photoresistor ADC value's lower limit.

When the application is launched, there may be some ambient light, to set the LED to off condition and turn the LED brightness. The line of code, "constrain(Ls_rsensorValue, Light_threshold, 1023)", is used to limit the ADC value from the photoresistor to be 1023 or lower, to avoid mapping the value outside of the 0~255 range. This is a little trick to work with sensor circuit, to enable the sensor circuit to function under different condition and establish a simple power-on calibration condition.

After deploying the sample application to EduCake, you can see the LED is in off or almost off condition when the ambient light remain constant. By covering the photoresistor senor, the LED brightness gradually increase and will reach maximum brightness level when the photoresistor is covered from the light source. From the Serial Monitor, you can see the ADC's changing value. By connecting to a larger light bulb (which require external power source), the circuit can be used as supplemental lighting that increase brightness as the ambient light decreases. When you turn the knob on the variable resistor to change it's resistance value, the servo will move relative to the movement on the variable resistor. When the value in the delay() function is increase, you can see the respond from the Servo is not smooth, which is caused by lowering the frequency to read and map ADC value from the variable resistor to the servo.

The 86Duino EduCake provides an easy to use environment to create different type of automation control application that interact with the surrounding environment. We will work on more advanced example in the later chapters.