PWM Tutorial

1. PWM Introduction

Short for Pulse Width Modulation, PWM is a technique to simulate analog signal from digital signal. PWM is commonly used for regulating voltage to control motor speed, lighting intensity, LCD backlight control, sound and audio frequency output signal and etc. PWM signal is generated by modulating a digital output signal on and off, usually incorporating a TIMER, to output a series of square wave, as shown in figure-1.

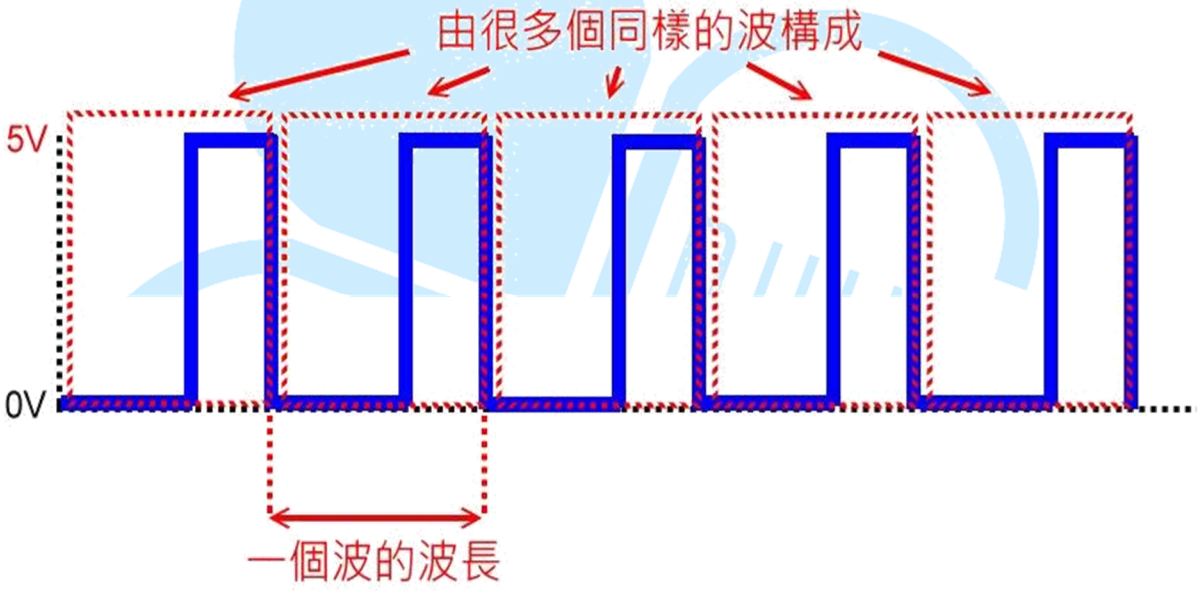

Figure-1. PWM signal

The length of time for each square wave is refer to as the duty cycle. Frequency (Hz) is the number of square wave over a period of time. For example, 60 square wave within a period of 1 second is 60 Hz/Sec. A continuing stream of these square wave, adjusted to different frequency and duty cycle, can be applied to different purpose, as shown in figure-2.

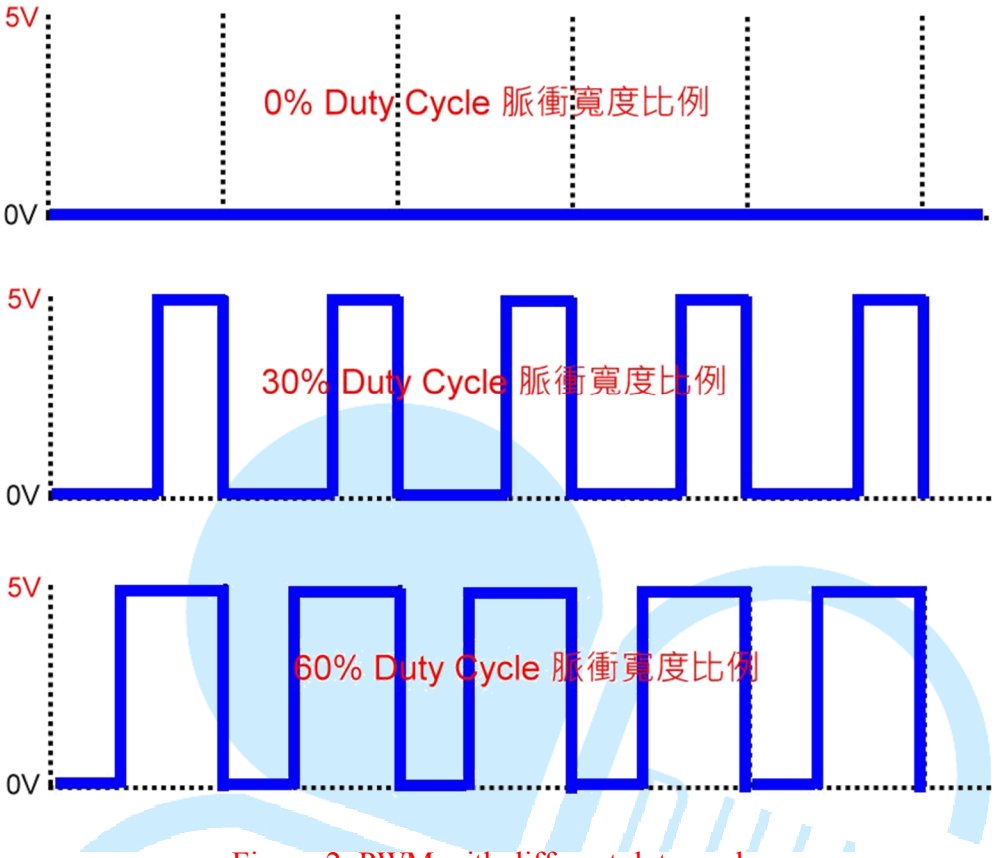

Figure-2. PWM with different duty cycle

You don't need to write complex program to implement simple PWM output from an 86Duino platform. The analogWrite () function is provided to generate PWM output, as follow:

- To generate PWM output as the 1st wave in figure-2 (0% duty cycle), use the following function:

  analogWrite (<pin ID>, 0);

- To generate PWM output as the 2nd wave in figure-2 (30% duty cycle), use the following function:

  analogWrite (<pin ID>, 78);

- To generate PWM output as the 3rd wave in figure-2 (60% duty cycle), use the following function:

  analogWrite (<pin ID>, 153);

Two parameters are needed when calling the analogWrite() function to output PWM signal, as follow:

```
analogWrite(Pin_ID, Value);
```

The range for Value is between 0~255, when mapping this range to 0~5V, the resolution is (5V)/255 = 0.0196V

One of the PWM output graph in figure-2 above is generated from the following function call:

analogWrite(Pin_ID, 160);

With a 5V digital output, the above function trigger a PWM output equivalent to 3.13V, (160 * 0.0196V), at 1 KHz frequency. The analogWrite() function can only be used to output PWM signal for I/O Pin on the 86Duino platform labeled with the "~" mark, as shown in figure-3.

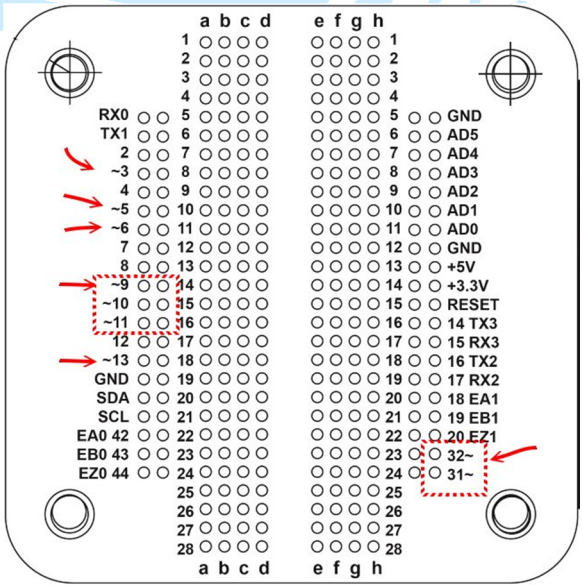

Figure-3. PWM capable Pin on the 86Duino EduCake

More importantly, since the output is generated from digital signal, by varying the time and frequency between the digital output's HIGH and LOW condition, the signal is not easily affected by interferences. Since PWM is an easy technique to implement, more and more applications adopted and use PWM for wide variety of application. One of the common and simple application for

PWM is controlling LED lighting, gradually increase and decrease the LED's lighting intensity, with codes such as the following:

```
digitalWrite(3,HIGH); // Set Pin-3 to HIGH
delay(1000);
digitalWrite(3,LOW); // Set Pin-3 to LOW
```

The above application codes control LED attached to Pin #3, turn the LED on, wait for one second and turn the LED off. Using digital output as the control signal, this is a simple control to blink the light between on and off status, with delay time in between.

We can use an analog output instead of the digital output, to create more variations and make things a whole lot more interesting.

Without PWM, analog output is more complex to generate comparing to digital signal. PWM signal provides a simple approach to generate analog output, as shown in the following sample codes:

```
// Code in this loop gradually increases LED brightness
for(int a=0;a<=255;a++) //
{analogWrite(3,a); delay(10);}
delay(1000);
// Code in this loop gradually decreases LED brightness
for(int a=255;a>=0;a--)
{analogWrite(3,a); delay(10);}
```

When the above codes execute, the LED brightness gradually increases, until it reaches the brightest level. Then, the LED brightness gradually decreases, until the light is completely off.

With some minor modification to the above code, we can create more interesting effect, such as controlling the LED to flicker like candle light. The author acquired a lamp for his bedroom, when the lamp is switched on, it

gradually increases brightness, then gradually decreases brightness and repeat the process. As the lamp repeat the process, the time for each brightness changing cycle become longer and longer, and eventually turn off and remain off after about half an hour. Similar control can be applied to control soft music melody, to gradually lower the volume in time, is an effective way to help small children to fall asleep. Obviously, application code to deliver behavior like this is more complex than the sample code above.

Using the above sample code as a starting point, instead of the a++ and a-- function to increase/decrease the value based on a fixed pattern, using a small range of random value during each iteration help create a more interesting flickering affect. With a sound sensor added, you can simulate the effect where you can blow some air to extinguish a real life candle, by detecting the noise level. Or, detecting the clapping sounds from your hands as the signal to control the lamp.

For the above LED application scenario, in addition to the application code, there are so electronic issues that affect the LED's behavior. For example, some LED requires 2.5V and higher to turn on. To simulate 2.5V and higher output from a PWM pin, the analogWrite (Pin_ID, Value) function must be call with 125 or higher as the Value. Within the PWM output's possible range from 0 to 255, the LED's brightness is not visible when the PWM output value is between the 0 to 124 ranges. When the voltage apply to LED between 4 to 5V, the change to the LED's brightness is not noticeable to human eyes. To create noticeable LED flickering effect, we need figure out the minimum voltage to turn on LED's brightness and the voltage where the LED's brightness is at the highest and apply the appropriate PWM value to the application code.

With basic information about PWM covered in the earlier section, let's take a look at PWM output for the 86Duino EduCake. The frequently for the EduCake's PWM output is 1000 Hz/Second and generate square wave output with 1/1000 second (0.001 second) width, which is sufficient for most application scenario (There are other options to product higher resolution PWM signal). Each of the PWM output from the 86Duino platform provides very low current. To support device the need higher current, such as motor and fan,

additional circuit and components to amplify the PWM signal are needed. In the following section, let's take a look at some PWM sample applications.

2. PWM Application 1: Flickering LED

This sample application involves a simple circuitry, with an LED and resistor. As shown in figure-4, the LED and resistor are connected in series and attached to two pins on the solder-less breadboard.

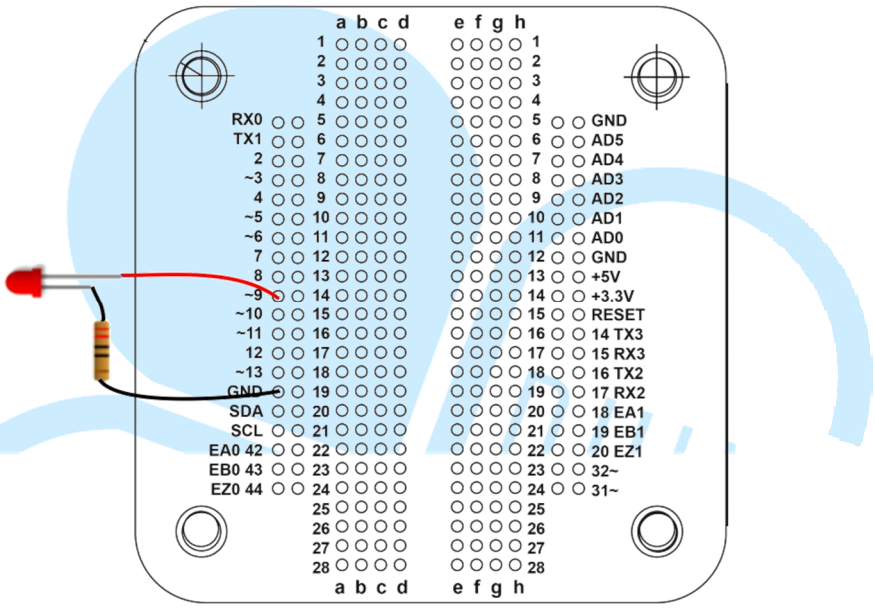

Figure-4. LED Circuit

Following is a listing of the sample codes for the application:

```
int brightness = 0; // brightness value
int fade= 5; // value to control change in brightness
int delaytime= 50; // value to control the rate of change
void setup()  {
  pinMode(9, OUTPUT);
}

void loop()  {
  analogWrite(9, brightness);
  brightness = brightness + fade ;
  if (brightness <= 50 || brightness >= 205) {
    fade = -fade;
  }
  delay(delaytime);
}
```

The two values in the "**if (brightness <= 50 || brightness >= 205)**" statement, '50' and '205', specify the range of value that control PWM signal output. For the LED used in this example, 50 is the minimum PWM value that turn on the LED's brightness to be visible. The LED will remain dark (turn-off) when the value is below 50. The LED reach its highest level of brightness when the control value for PWM reach 205 and higher. The range of value for PWM control signal, to control the LED brightness between minimum (off) and maximum brightness is different between different LEDs. You need to launch some test codes with different value to find the range suitable for the LED you are using. Without using these value to limit the range of value that produce visible change, the change in the LED's brightness will pause when the value is less than 50 and higher than 205.

How can we create a flickering LED to simulate candlelight? First, let's look at how candlelight behave:

1. Flame from the candle always emit light.
2. Airflow movement causes the flame from the candle to move and flicker. To fully simulate this behavior, we can use a servo motor to move the LED. Since we are using a simple LED circuit in this example, we will skip the servo motor usage in the later section. Another simple method to simulate flickering candlelight to place a sheet of transparent cellophane over the LED and use a small fan to generate airflow against the cellophane.
3. Airflow movement causes the candlelight flicker and change the illuminating brightness randomly.

While the above example is simple to some reader, the best approach to solve a problem or learn new technology is to start with the basic, and break apart a large complex problem into a group of smaller individual components, that can be easily transform into programming routines, where the results from each of these components can be combined to form the result and answer to the larger complex problem. Dissecting a large complex problem into a group of smaller and simpler problems, transforming these smaller problems into separate

programming routines and combine the results from these routines to form the answer to the large complex problem is an effective and pragmatic approach to solve big and complex problem.

Next, let's take a look at the following codes:

```
int brightness = 160;
int fade;
int delaytime= 20;

void setup()  {
  pinMode(9, OUTPUT);

  analogWrite(9, brightness); // 1st behavior · candlelight always on

randomSeed(analogRead(0)); //Random value for the 3rd behavior
}

void loop()  {
  fade= random(0, 61); // create random number between 0~60
  fade-= 30 // subtract 3 to adjust the range to -30 ~ +30


// 3rd behavior, random brightness
   brightness = brightness + fade ;
// fade is a random value, to generate flickering effect
// Set the lower limit to '60', preventing complete darkness.
  if (brightness <= 60 )
     brightness = 60;


// Set the upper limit to '200'
  if (brightness >= 200)
     brightness = 200;
     analogWrite(9, brightness);
    delay(delaytime);
       }
```

With some tuning, the value and parameter in the above codes can adjusted to deliver good result. When using RGB 3 colors LED with the above example, it generates interesting result.

3. PWM Application 2: Servo Control

PWM is a common signal to control servo. The PWM output from the EduCake can be used to control servo motor commonly used in the remote control application, such as the SG90, an entry level servo as shown in figure-5.

Figure-5: A servo with common 3-pin interface

This type of low cost servo is common in the remote control market, available for purchase from various outlet, including online ecommerce site. The 3-pin connector provides the interface to connect to Power, Ground and PWM signal. For the SG90 servo in figure-5, the red-wire is power, brown-wire is ground and orange-wire is the PWM signal, which can connect directly to the PWM output from the EduCake. Some servo manufacturer use different color of wires for PWM signal.

Although this type of entry level servo has limited torque, speed and accuracy, it's good for learning and practices.

Here are the specifications for the SG90:

1. Weight: 9.0 g (0.32 oz)
2. Dimension: 23 x 12.2 x 29 mm (0.91 x 0.48 x 1.14 in.)
3. Torque (at 4.8V): 1.8 kg/cm (25.0 oz/in)
4. Speed (at 4.8V): 0.1 sec/ 60degree.
5. Dead band width: 10us (Dead band width is an interval between PWM pulses. Where PWM pulse false within the dead band width zone, the pulse does not affect servo movement.)

6. Max travel: 150 degree (**Important: PWM signal attempting to move the servo beyond its range can overload and damage the servo.)

There are other similar servo built with higher torque, faster, smaller dead band width to support project that needs higher performance servo, such as the RS-0263 in the following URL:

http://www.roboard.com/Files/RS-0263/RoBoard_RS-0263.pdf

As shown in figure-6 below, a simple circuit with the SG90 servo directly connected to the EduCake's I/O. While the SG90 servo draw very little current, making it possible to directly connected to the controller, it's not a good practice. Comparing to the controller board's I/O, servo draws higher current. The servo should be connected to a separate power source, where the GND for the separate power source and the controller board are linked, to prevent the servo from drawing too much current and affect the controller board's function. In addition, diodes can be added to provide protection against reverse current.

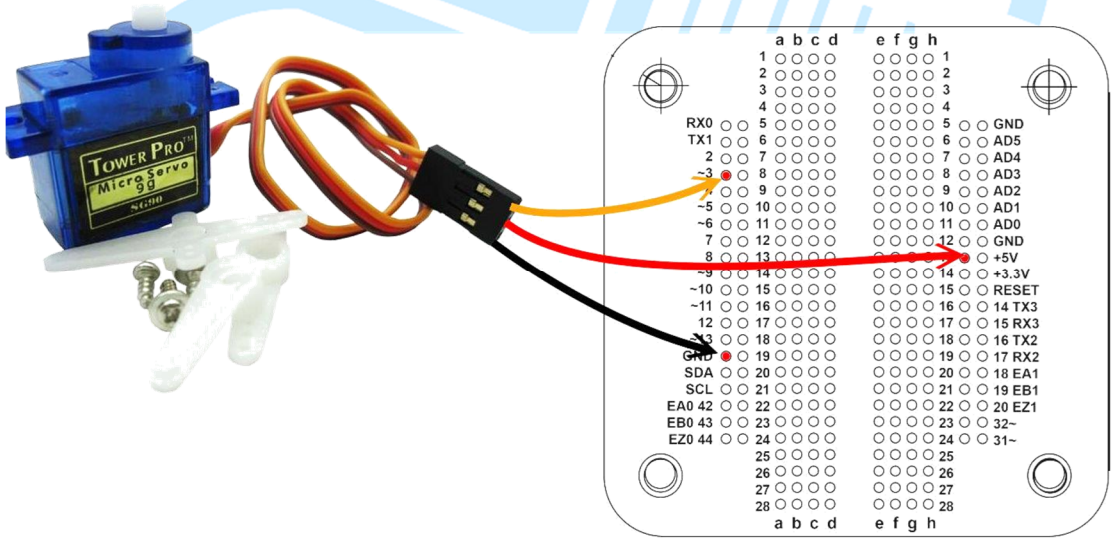

Figure-6: Connecting Servo to EduCake

There are other similar servo built with higher torque, faster, smaller dead band width to support project that needs higher performance servo, such as the RS-0263 in the following URL:

The source code in the next listing output control signal to move the servo arm back and forward. The author uses similar code to test the servo, with additional weight attached to the servo arm and let the servo run until it dies, to

test its reliability, torque, current consumption and behavior under different operating temperature, to validate whether the servo meet the requirements for the intended project. When selecting a servo, in addition to looking at the common servo functional specification, the servo's dead band width is an important factor that affect the servo's performance and accuracy, servo with smaller dead band width is better.

```
#include <Servo.h>
Servo myservo; // create Servo object
void setup()
{
  myservo.attach(3);    // Associate myservo to digital pin #3
}
void loop()
{
  for(int a = 30; a <= 150; a++){
    myservo.write(a); // Move servo between 30~150 degree
    delay(20);
  }
  for(int a = 150; a >= 30; a--){
    myservo.write(a); // Move servo between 30~150 degree
   delay(20);
  }
}
```

In the above source code listing, in addition to the comments, there are a few things you need to pay attention to. The codes in the above listing limit the servo movement between the ranges of 30 to 150 degree, within the SG90 servo's operating range.

When using value from a variable resistor to control servo movement and synchronizing the servo's position with the dial on the variable resistor, it's important to consider the following:

- Most of the variable resistor can rotate within 120 degree range
- The range of movement for the servo vary between different manufacturers, some servo is designed with 80 degree movement (40 degree each direction away from the center), some servo is designed to support 360 degree movement. When sending control signal to move the servo beyond its supported range, it can overload the internal circuit and

cause permanent damage to the servo. It's important not to exceed the servo's supported range of movement.

- When using servo without datasheet, and the vendor not able to provide technical specification for the servo, you can test the servo using the above source code, starting with a smaller range of movement to check whether the servo is able to complete the movement smoothly. For example, instead of the 30 ~ 150 range, expand the range to 25 ~ 155, if the servo is able to complete the movement, increase the range of movement by a small value and test again (such as 20 ~ 160), until it reaches a set of value where the servo reach it's max movement range and generate noticeable noise while it attempts to move beyond the supported range. When reaching this point, you need to quickly stop the code from running or remove power from the servo. Otherwise, it can damage the servo.

- Another item of concern is the line of code "delay(20);", which relate to certain servo behavior. The common PWM pulse supported by the low cost servo is 50 Hz/sec, which produce a 20ms pulse (1second/50 = 20ms). As the result, after the current PWM pulse changed the servo position, the servo is not able to response to change until the next pulse, in 20ms. When writing a program that frequently change a servo's movement and position, sending multiple command to change the servo movement within the 20ms time slot does not help and my interfere with the intended servo movement. It's best to limit the number of servo movement command being send to the servo to 1 per PWM pulse, in this case 20ms.

- There are higher performance servos in the market that support 65Hz, 120Hz, 200Hz, 333Hz and faster pulse, which enable these servos to be more responsive and able to move faster. Some high performance servo, such as the one from KONDO, is built to support 300us PWM pulse and able to provide feedback with information that indicates the servo's current position. However, the advanced subject is beyond the scope for this chapter.

4. PWM Application 3: DC Motor Control

DC motors have simple design, inexpensive and can easily purchase through electronics store and online ecommerce website. You can find DC motors in different type of application, including toys, electric fans, electric bicycles, electric gate and others. Current required by most DC motors is higher than the current able to supply by the EduCake's I/O and cannot be connected directly to the EduCake.

One of the simple and low cost solution to drive a DC motor using PWM output from the EduCake (and other controller) is to implement a current amplification circuit, using 9012/9013, 2N2222/2N2907 or TIP120 transistors. Implementing a current amplification circuit with transistor requires electronics knowledge to calculate and select proper resistors, diodes (to prevent reverse current) and develop the necessary schematic to implement on a breadboard for testing or create a PCB. Without electronics background, you need to invest times and efforts to understand and carefully put together the current amplification circuit and connect to the EduCake to make the DC motor move. To the experienced DIY hacker, this is a simple subject.

A simpler alternative to creating your own current amplification circuit is to use an integrated circuit (IC) chip, designed to provide current amplification, such as the L293 IC chip. It's very simple to create DC motor control circuit using L293 IC chip. As shown in figure-7, two DC motors, such as those in electronic toys, can be connected to an L293 chip. The circuit diagram in figure-7 only shows the connections to the DC motors, the other pins on the L293 IC are used to control the motors to move forward, reverse and speed control. To learn more about L293 IC, refer to the datasheet on the following links:

http://pdf.datasheetcatalog.com/datasheets/90/69628_DS.pdf

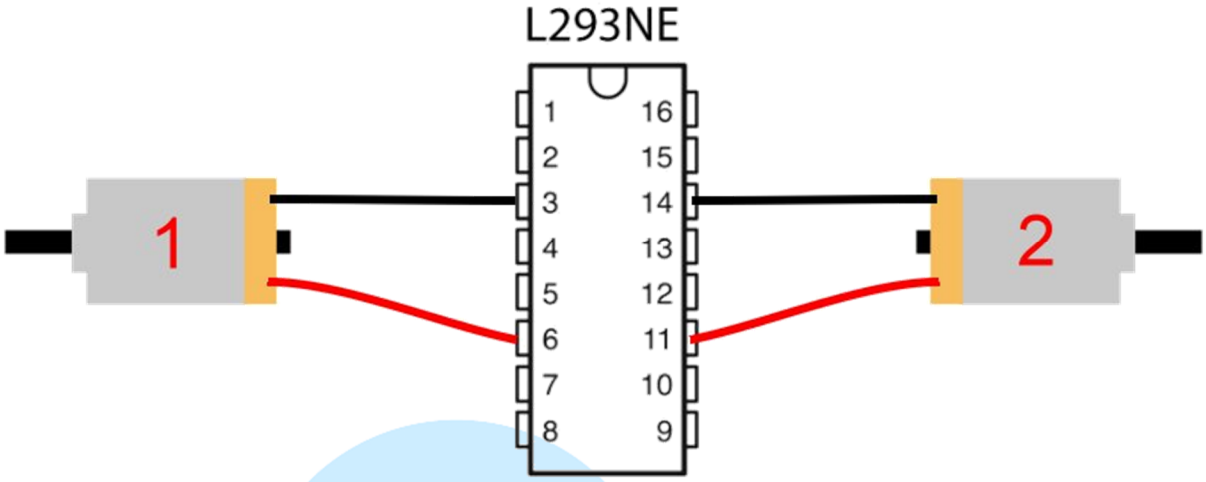

Figure-7: DC motors control using L293 IC

The following table provide function description for the L293 IC, as shown in Figure-7:

Pin	Description
4、5、12、13	GND
1、9	Speed control for motor 1 and 2.
8	Voltage supply for the motors, supports 4.5V to 36V range. Use PWM to supply voltage below 4.5V. Note: Output to motor max out at 0.6A. Avoid using motor that draws more than 0.6A.
16	VCC (Typically 5V)
2、7	Motor 1 movement control (forward, reverse & stop)
10、15	Motor 2 movement control (forward, reverse & stop)

Using information from the above table, we can implement a simple DC motor control circuit using an L29 IC and the EduCake, as shown in figure-8. To keep the example simple and easy to understand, one motor is used. By using a small DC motor from electrics toys that draw small amount of current (in the range of 15-20 mA), the 5V supply from the EduCake provide sufficient current to drive the low current small motor. When using a motor that draw higher current, to avoid damage to the EduCake, use an alternative power source for the motor.

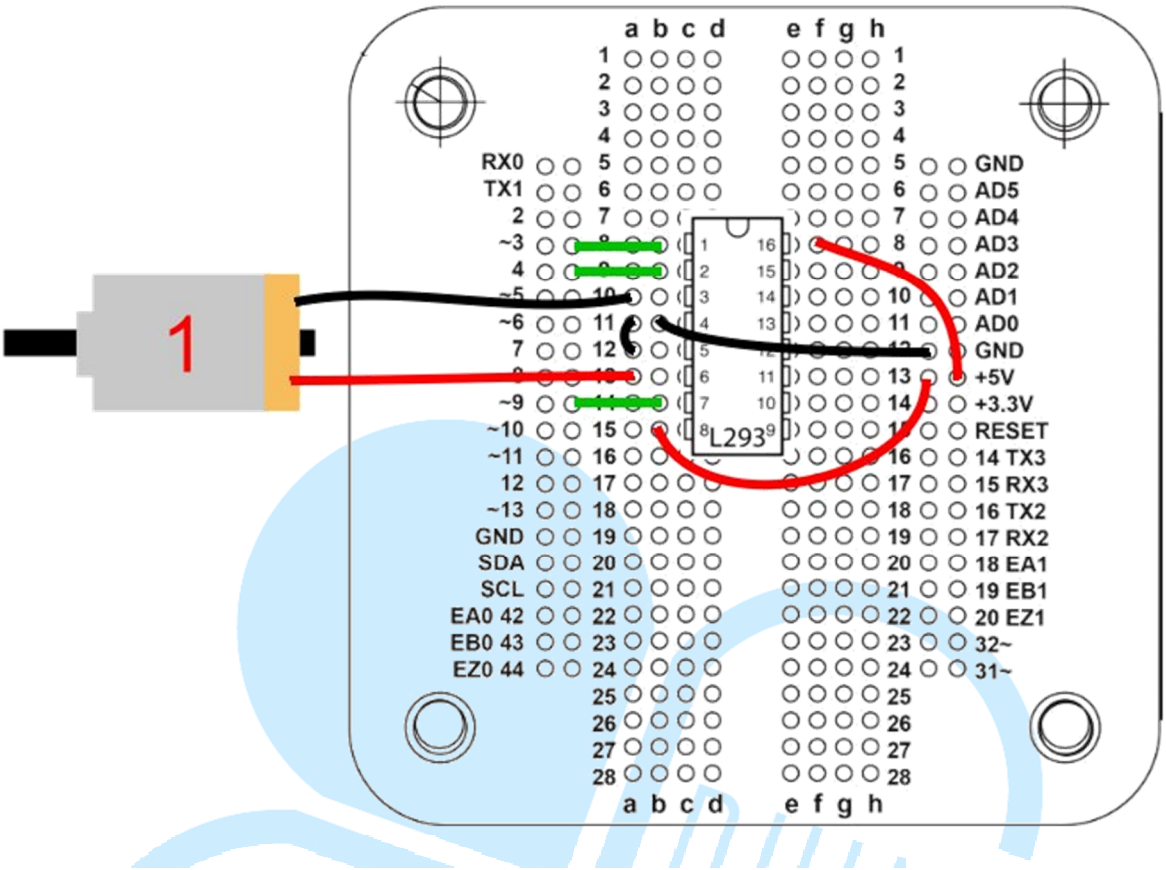

Figure-8: DC motor control circuit using L293

The application code in the following listing send control signal to the DC motor to perform the following:

- Turn the motor at maximum speed in one direction for 1 second and stop.
- Turn the motor at maximum speed in reverse direction for 1 second and stop.
- Slowly accelerate the motor from stop to maximum speed, and stop.
- Repeat the above steps.

```
int M1=4,M2=9; // Can connect to any 2 digital output
int EN=3; // Must connect to a PWM capable output

void setup()
{
   pinMode(M1,OUTPUT);
    pinMode(M2,OUTPUT);
    pinMode(EN,OUTPUT);
}

void loop()
{
   int a;
    digitalWrite(M1,HIGH); //Turn motor in one direction
    digitalWrite(M2,LOW);
    digitalWrite(EN, HIGH);
    delay(1000);

    digitalWrite(EN, LOW); //Stop
    delay(1000);
    digitalWrite(M1,LOW); //Turn motor in reverse direction
    digitalWrite(M2, HIGH);
    digitalWrite(EN, HIGH);
    delay(1000);
    digitalWrite(EN, LOW); //Stop
    delay(1000);

    for(a = 0; a <= 255; a++)
    {
     digitalWrite(M1,HIGH);
        digitalWrite(M2, LOW);
        analogWrite(EN, a); // Sent PWN signal to EN pin, to
                                        // control motor speed
        delay(20)
    }

    digitalWrite(EN, LOW); // Stop

    delay(1000);

}
```

```
analogWrite(EN, a);      // Sent PWN signal to EN pin, to
                         // control motor speed
```

In the above line of code, a command to output PWM signal to control motor speed, is the same as the code to control LED brightness in the earlier section. Similar to the LED, PWM voltage intensity below certain level is not sufficient to cause the motor to turn. The required minimum value to move the motor vary between different types of motor. Even the same model of motor may require slightly different minimum voltage value to move.

To find out the minimum value needed to move the motor, you need to run a series of tests by sending PWM control signal, starting with a small value, and gradually increase the value until the motor starts to move, as follow:

- To start with sending PWM signal equivalent to 0.5V. We need to calculate the value for a, using the following formula:

  (0.5V / 5.0V) * 255 = 25

  To output PWM signal equivalent to 0.5V, replace 'a' in the above line of code with 25.

- Repeat the above step with higher value for 'a' until the motor starts to move.

The motor control circuit in figure-8 is connected to a 5V power source. When working with 3V motor, you need to limit the PWM output to 3V to avoid damaging the motor. (3V / 5V) * 255 = 153, the maximum value you can set for 'a" in the PWM output command is 153. You can probably output PWM signal higher than 3V without immediately damage the 3V motor. Prolong overvoltage will shorten the motor's usable life span. Excessive overvoltage will eventually damage the motor, permanently. It's good practice to avoid overvoltage.

Using similar circuit and application control code, you can control much larger motor by changing the control IC that can handle higher voltage and current. Or, use an alternative power MOSFET circuitry (The author used

similar circuitry and application codes to control motor application for a two tons moving truck). When working with larger motor with higher current draw, be sure to use higher gage wire to avoid damage to smaller gage wire.

5. PWM Application 4: Brushless DC Motor Control

In addition to the things covered in the earlier section, the author used PWM for different application, including remote control airplane, boat and other automation application. However, most of these applications do not use DC motor. DC motor is not designed to deliver high torque, not responsive to change its rotating speed quickly and not able to support application that require much high rotating speed in the range of 10K~20K RPM or higher. For high rotation speed application, it increases the wear to the brush and contact area inside the DC motor, damage these mechanical components and shorten the motor's usable life span. In addition, the brush in a DC motor can generate spark at high speed, a serious problem for many application.

In the remote control market, brushless DC motor are used to avoid the short coming associate with brushed DC motor, such as the one shown in figure-9.

Figure-9. Brushless DC motor

The brushless DC motor shown in figure-9 is built with much higher quality and performance comparing to the typical low-cost DC motor found in the toy market. The cost for this type of motor is also significantly higher.

Notice: The interface to motor in figure-9 includes 3 wires.

To control brushless DC motor using EduCake, we need to understand the motor's behavior, performance and power consumption. When looking at the specification for a motor, the KV value represent rotation speed, referring to the number of revolutions the motor turns per minute when 1 Volt is applied without any load.

The requirements for motor in term of speed and torque are not the same for different type of application, such as remote control airplane, car, boat and etc. There are broad range motors with different KV value available in the market to support different type of applications.

Depending on the application, you need to select a suitable brushless DC motor. For example, when working with a remote control airplane, the size of the airplane affects the required size for propeller and KV value for the motor to provide sufficient propulsion to lift the airplane. Propulsion force is relative to propeller size and rotation speed. A larger size propeller rotating at slower speed can deliver the same propulsion force as a smaller size propeller rotating at a higher speed. The required KV value for a motor to lift the airplane is dependent on propeller size and airplane's weight. Larger size propeller also requires motor with higher torque.

It's important to know a brushless DC motor can draw very high current and generate high heat. Without proper precaution, the high current can easily burn out wiring harness not intended to support high current. The high heat can cause fire hazard under certain condition.

Many remote control applications include an electronic speed control (ESC) module as part of the brushless DC motor control circuitry. The ESC module includes combination of regulator and control circuit to simplify motor control. The circuitry for ESC is not difficult to understand. However, ESC is not part of the objective for this application note. We will use an off-the-shelf ESC module instead, as shown in figure-10.

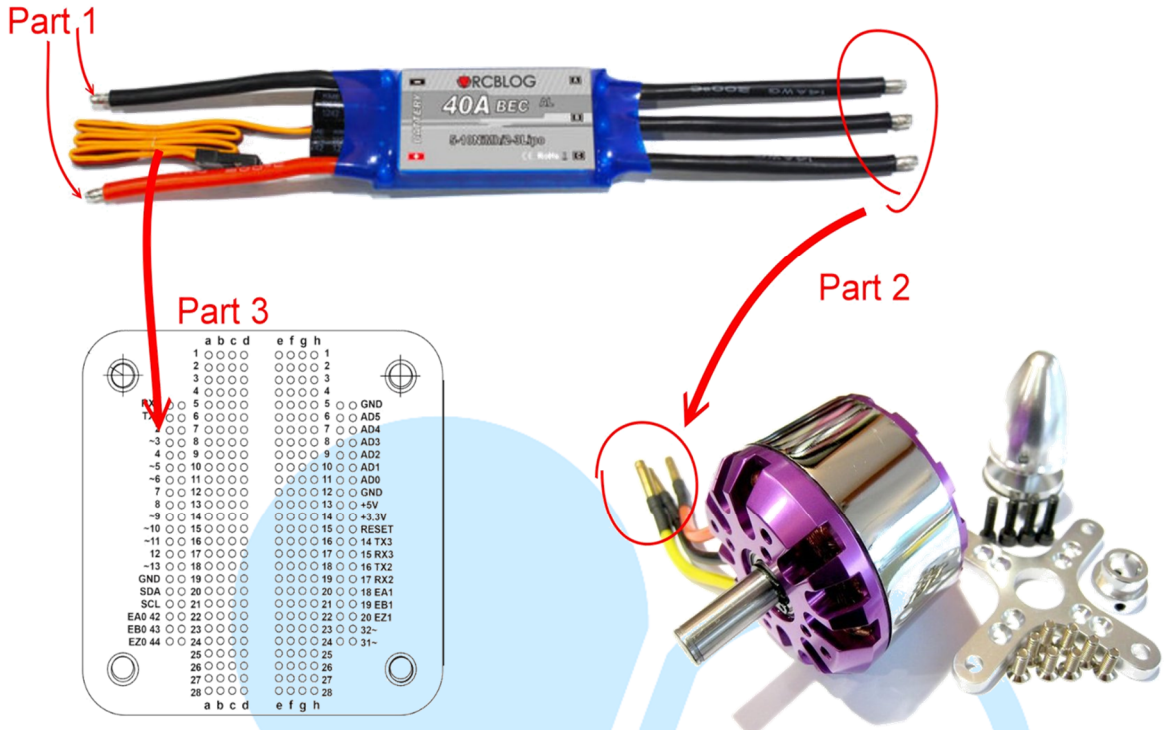

Figure-10. ESC and Brushless DC motor

In the following section, we will talk about putting together a circuitry with brushless DC motor, ESC and EduCake as the controller. The circuit is separate into 3 parts, part-1 power to the ESC, part-2 connection between ESC and the motor and part-3 PWM output from EduCake to the ESC module.

Part1- For this brushless DC motor control circuit, external power to drive the motor is connected to the ESC module. It's important to connect the power source to the ESC module correct to avoid reversing the polarity. Incorrect power connection can damage the circuit. It's common for ESC module to support 7.4V to 11.2V input voltage range. DC motor can draw extremely high current. Make sure the battery, or power supply, can deliver sufficient current needed drive the motor. Check to make sure the wires and connectors can support the anticipated high current.

Part2- There are three wires from the ESC module that need to connect to the three wires on the brushless DC motor. The three black wires from the ESC module are marked with A, B, and C without indicating which wire is positive or negative. When applying power to brushless DC motor, reversing the power

connections' polarity simply reverse the motor's rotating direction. Apply power to test the motor's rotating direction. To change motor rotating direction, keep the center wires connected and reverse connection for the outer two wires. For remote control application, such as the quad copter, the 4 brushless DC motors are configured as 2 separate pair of motors, where the motors for each pair are configured to rotate in the appositive direction from the other pair, to help maintain balance. If the motor for your application is not rotating in the expected rotation direction, simply reverse connection for the two outside wires.

Part 3- The PWM output from the EduCake is connected to the ESC module's control signal wire, the orange wire in the middle on the input side. The signal level for the control signal wire is the same as the signal level used to control Servo motor.

The above brushless DC motor control circuit involve high current, which can permanently damage the affected components when power connections are connected incorrectly. Check to insure all connections are correct before applying power to the circuit.

In the next section, we are going to write some application codes to control the motor. While the codes needed to control the brushless DC motor is quite similar to the codes to control SERVO, the ESC module's behavior is different from a SERVO, which we need to pay attention to. When using an ESC module, it needs to be calibrated first.

Following are the typical steps to calibrate an ESC module for a remote control airplane:

1. First, turn on the remote control and move the throttle control to the maximum position. Then, apply power to the receiver.

2. The ESC module emit a beep. At this point, move the throttle control to the minimum position, to capture the maximum to minimum throttle movement.

3. At this point the ESC module emit a beep sound again, to indicate the calibration is completed.

4. Some ESC module is able to save the calibration setting, to save you from the hassles of calibrating the module frequently. Many ESC module is not able to save calibration setting and have to be calibrated every time.

Most of the ESC module in the market is designed to use similar calibration steps as above. However, there are different calibration steps for ESC module from different manufactures. You need to refer to the ESC module user manual to find accurate steps. During the calibration process, the motor may receive control signal to rotate at maximum speed, which can be a safety problem for high speed motor. To prevent problem, remove the motor from the circuit while testing the ESC module calibration process.

After you are able to validate the steps needed to calibrate the ESC module, let's take a look at the following application codes:

```
#include <Servo.h>
Servo myservo; // Create the Servo object
void setup()

{
  myservo.attach(3);    // Initiate myservo to use pin #3

}
void loop()
{
 int a = analogRead(0); // Read value from analog Pin #0
                                          // Variable resistor is attached to
                                          // Pin #0 to control speed


 a=map(a,0,1024,1000,2100); // Translate the speed control
                                                  //    value to PWM

 myservo. writeMicroseconds (a);    // Output speed control
                                                          // in microseconds to
improve
                                                          // control precision.

    delay(20);

}
```

In the above codes, "**a=map(a,0,1024,1000,2100);**" is reading value from the variable resistor, which correspond to PWM control signal in the range from 1000 to 2100 micro-second. Then, use the "**writeMicroseconds()**" function to output PWM signal to control the motor.

The "**writeMicroseconds()**" function can output much higher resolution control signal than the "**write()**" function. There are many different type of SERVO with different sensitivity to PWM control signal. The "**write()**" function, which has the much lower PWM output resolution, is not able to control many of these SERVO effectively. The author uses the "**writeMicroseconds()**" function, as the default for all SERVO control application.

After launching the above application code on the EduCake, you can turn the variable resistor to different position, to see the motor responses. Instead of the remote controller, you can use EduCake to create many different variable of control to support different type of application.