Pulse Width Modulation Using an Arduino



Travis Meade 11/14/2014

Abstract

Pulse Width Modulation is a process that is used in many applications. One of the easiest ways to implement this is by using an Arduino. The Arduino can do this in a number of ways. This application note will look at what Pulse Width Modulation is and will also explain how to perform Pulse Width Modulation using two different methods.

Keywords

Pulse Width Modulation, Arduino, AnalogWrite, DigitalWrite

Table of Contents

Abstract	1
Introduction	3
Objective	4
Methods	4
AnalogWrite	4
DigitalWrite	6
Conclusion	7
Reference	8

Introduction

To understand how to implement Pulse Width Modulation, it is first necessary to understand what it does. Pulse Width Modulation (PWM) is a way to provide a variable signal from a given set signal. PWM does this by changing the pulse width, which in turn, changes the duty cycle of a square wave to alter how much power is supplied to the attached component. It varies because the signal takes the duty cycle and averages the power signal that is output. This can be seen in Figure 1 below.

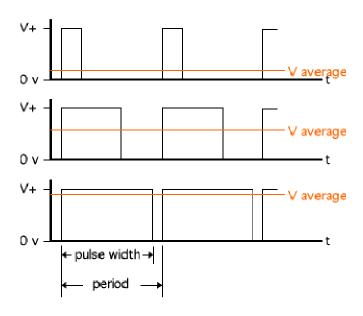


Figure 1. Varying Duty cycle [2]

So the larger the duty cycle percentage, the larger the output signal will be. In the picture above and for the Arduino, it is the voltage that varies.

The Arduino should also be briefly explained before moving on to how to create the PWM. There are many variations of Arduinos on the market. The Arduino Uno will be the focus of this application note. The Arduino is a board that allows the user to access, program and build with a microcontroller. The Arduino has 14 digital input/output pins [1]. Six of these pins can be used for with the PWM function that is already built into the Arduino. The pins that do not have this ability can still be used for PWM but must be used in another way. They output either a High (5V) or Low (0V) depending on what is coded. The coding will be explained in the sections that describe the different methods for PWM.

Objective

The objective of this application note is to provide an understanding of how to create a PWM signal using multiple methods on the Arduino Uno.

Methods

This application note will talk about the two most basic ways of implementing PWM. These methods are analogWrite and digitalWrite.

AnalogWrite

The Arduino has six pins that can already do PWM without varying much. Some sample code is provided in Figure 2 below.

```
int led = 9;
                     // the pin that the LED is attached to
int brightness = 0;  // how bright the LED is
int fadeAmount = 5; // how many points to fade the LED by
// the setup routine runs once when you press reset:
void setup() {
 // declare pin 9 to be an output:
 pinMode(led, OUTPUT);
// the loop routine runs over and over again forever:
void loop() {
  // set the brightness of pin 9:
  analogWrite(led, brightness);
  // change the brightness for next time through the loop:
  brightness = brightness + fadeAmount;
  // reverse the direction of the fading at the ends of the fade:
  if (brightness == 0 || brightness == 255) {
    fadeAmount = -fadeAmount ;
  // wait for 30 milliseconds to see the dimming effect
  delay(30);
}
```

Figure 2. analogWrite

The code is broken down into three sections. The first section is where the variables that are going to be used are initialized. For instance, the line that says "int led = 9;" sets pin 9 on the Arduino to the variable "led". The variable "led" can now be used in the code and it will tell the

Arduino that it is using pin 9. This makes the code cleaner and it helps keep track what each line in the code is actually affecting. The next section of the code is the void setup. This section is where pins are set as either inputs or outputs. This is done with the pinMode function. The function takes your pin or variable and then sets it to what you specify, either input or output. In this example, the variable "led" is being set as an output. The last section of code is the void loop. This is where the bulk of the code goes. It will run an infinite loop of the code that is put in this section.

AnalogWrite function generates a square wave that can be varied in the function. The two things the function needs to run are a pin/variable and a value to set the duty cycle. The value that gets input for this function has to be between 0 and 255 [3]. A duty cycle of 100% occurs if the value is set at 255 and a value of 0 gives a duty cycle of 0%. To get a specific duty cycle a value for analogWrite needs to be calculated. This is shown below.

$$duty\ cycle * 255 = analogWrite\ value$$

For example, a duty cycle of 20% would need to be set at a value of 51. The example code in Figure 2 has code that changes the duty cycle while it is running. The output of this code is shown in Figures 3, 4, 5 and 6 below. Each figure shows a different time. So the pictures from left to right shows the code changing the pulse width.

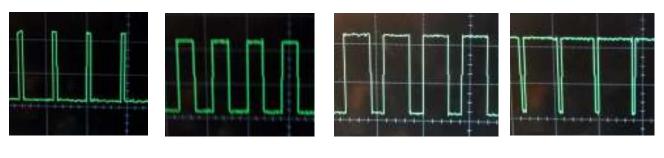


Figure 3. Figure 4. Figure 5. Figure 6.

DigitalWrite

The digitalWrite function can be used on all 14 digital input/output pins. Unlike analogWrite, a square wave has to be made manually. This is done by using both the digitalWrite and delay functions. The digitalWrite function can only provide a High (5V) or a Low (0V) [4]. So manually doing this in the code provides a square wave. Sample code is shown in Figure 7 below.

Figure 7. digitalWrite

Each time the digitalWrite function is used, it needs to be followed with the delay function. The above code currently has a duty cycle of 50%. To get a desired duty cycle, the math becomes a little harder than for the analogWrite function.

$$\frac{delay\ time\ HIGH}{delay\ time\ HIGH+delay\ time\ LOW} = Desired\ duty\ cycle$$

Using the same example as before, a 20% duty cycle could be implemented using a delay time HIGH as 400 and a delay time LOW as 1600. The good thing about the digitalWrite function is that frequency can also be changed. Frequency is the inverse of the period or total time it takes for one cycle. So to change frequency, the total time just needs to be increased or decreased. The current example takes 2 seconds so the frequency is 0.5 Hz.

Conclusion

The two methods have their pros and cons. AnalogWrite already generates a square wave and it is simple to calculate the value needed to get a desired duty cycle. The AnalogWrite function cannot have a different frequency than what is already set for that pin. The DigitalWrite function requires more math to get the desired duty cycle but it allows for the frequency to be changed. More information about both of these functions can be found on the Arduino website [5]. Arduino is an open source microcontroller board so people are constantly writing new code to be implemented on these boards. Other methods of generating a PWM signal are available online but these are the most reliable because they were made available with the Arduino board.

References

- [1] http://arduino.cc/en/Main/ArduinoBoardUno
- [2] http://motorcontroller.blogspot.com/2009/07/today-i-looked-over-my-ece-618-junior.html
- [3] http://arduino.cc/en/Reference/analogWrite
- [4] http://arduino.cc/en/Reference/digitalWrite
- [5] http://arduino.cc/