EEE-201 Chapter 2

Now that we know what sort of signal a servo requires in order to be controlled, let's learn how to do that with an Arduino. We'll also be looking into how to make this process a lot easier through the use of the Servo library; and we'll be diving into more programming by learning several new control structures.

Table of contents

Part 1 - square wave on the Arduino

We saw previously that a square if simply a wave that spends part of its time in the `HIGH` (or `ON`) position, and part of its time in the `LOW` (or `OFF`) position; we also saw that this square wave can be used to drive a servo. So, how can we do this with the Arduino? Download the sketch SquareWave.ino and find out!

The sketch is quite basic, with all the magic happening in the `loop()`:

```
void loop() {

  // turn pulse on for pulsewidth duration
  digitalWrite(pin, HIGH);
  delayMicroseconds(pulsewidth);

  // turn pulse off
```

```
    digitalWrite(pin, LOW);
    delayMicroseconds(period - pulsewidth);

  }
```

All we're doing here is turning the servo pin `HIGH` for a duration equal to the variable `pulsewidth`, after which we turn the pin `LOW` for a duration of the desired `period` minus the `pulsewidth` (this way the total `HIGH`/`LOW` time is equal to `period`).

If we look on the oscilloscope, we can see the clear shape of the square wave, and we can see that the wave is `ON` for the duration of `pulsewidth`. However, the keen observer should have noticed that the period isn't 25ms. The reason for this is a bit technical but part of it has to do with the fact that behind the scenes, various other pieces of code are being run; also, the function `digitalWrite()` is relatively slow. All of this works together to throw the timing that we are hoping for (in this case a square wave of 50 Hz) off. One way of getting around this is to use libraries!

Part 2 - Arduino libraries

One way of getting around the messy timing of signals and all the lines needed just to drive the servo are to use Arduino libraries

> The Arduino environment can be extended through the use of libraries, just like most programming platforms. Libraries provide extra functionality for use in sketches, e.g. working with hardware or manipulating data.

You can think of libraries as snippets of code that make doing things like controlling servos or working with other hardware much easier. Let's

take a look at the Servo library by loading the Servo.ino sketch.

Let's break the code down.

```
#include <Servo.h> // load the servo library

Servo panServo;  // create servo object to control a servo
```

We begin by including the Servo library into our sketch so that the computer knows it has to upload that Servo library code to the Arduino. Then, we create a new `Servo` object and name it `panServo` - in programming speak we *instantiate* the object. You can think of this as your physical servo; if you had two servo, then you would *instantiate* two `Servo` objects.

We then use the method `attach()` to assign the control pin to our `panServo` through the code `servo.attach(10);` - in this case we are using digital pin 10.

Then, within the loop, we tell the servo to go various positions through the line of code: `servo.write(0);`

One thing you should notice when using libraries is that each library will allow you to use special methods (or functions) as defined within that library. In our case, we have access to methods such as `attach()` and `write()`. If you consult the documentation you can see there are many more functions available to the `Servo` object.

Part 3 - control structures

Now that we can easily control the position of servos, let's learn about a few more control structures to make our coding even more powerful.

Load the Sweep.ino sketch and plug in your servo.

The section of code that's most interesting is the the `for` loop and looks like:

```
for (pos = 0; pos <= 180; pos += 1) { // goes from 0 degrees to 18
  // in steps of 1 degree
  panServo.write(pos);              // tell servo to go to positio
  delay(15);                        // waits 15ms for the servo to
}
```

> A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

The for loop has three parts to it:

```
for ( init; condition; increment ) {
   statement(s);
}
```

1. *init* declares and initializes our loop control variable; in this case `pos` is set to 0.
2. *condition* is evaluated at each iteration of the loop. If `true` the body of the loop is executed; if `false`, the loop is exited.
3. *increment* changes the value of the loop control variable (`pos` in our case) every time the loop iterates. Thus, in our example `pos` is increased by a value of 1.

Through the use of for loops, we can change the position of the servo smoothly and create a repetitive back and forth motion.

Another way of achieving the same motion is through a `while` loop; the `loop()` below achieves the same servo motion as before:

```
void loop() {
    while (pos < 180) {
        pos += 1;
        myservo.write(pos);              // tell servo to go to po
        delay(15);                       // waits 15ms for the ser
      }
      while (pos > 0) {
        pos -= 1;
        myservo.write(pos);              // tell servo to go to po
        delay(15);                       // waits 15ms for the ser
      }
}
```

Like the `for` loop, a `while` loop checks whether a *condition* is true; as long as it is, the loop continues. You can read a great discussion about what the difference is between a for or while loop.

A discussion about loops would not be complete without discussing loop control structures: `break` and `continue`:

`break`:

> Terminates the loop or switch statement and transfers execution to the statement immediately following the loop or switch.

`continue`:

> Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.