LAB 4

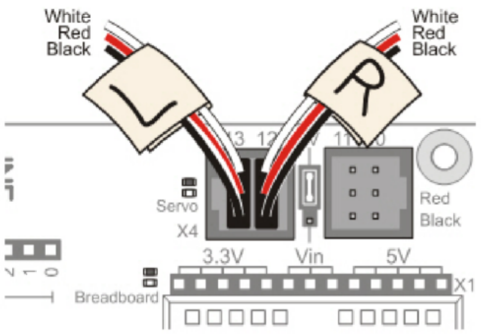
Attach the BOE Shield to the Chassis

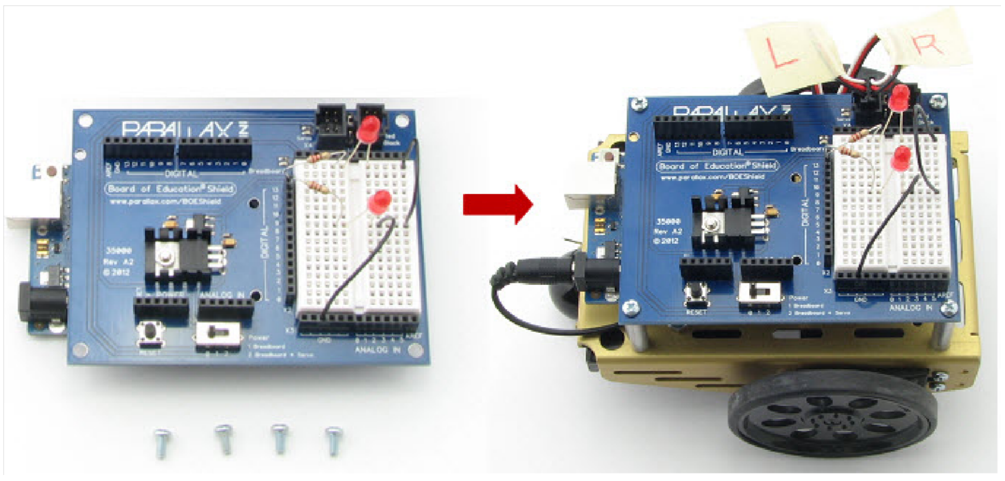
Parts List

(4) pan-head screws, 1/4″ 4-40
(1) Board of Education Shield mounted to your Arduino module and secured with standoffs

Instructions

- ✓ Set the BOE Shield on the four standoffs, lining them up with the four mounting holes on the outer corner of the board.
- ✓ Make sure the white breadboard is closer to the drive wheels, not the tail wheel.
- ✓ Attach the board to the standoffs with the pan head screws.
- ✓ Reconnect the servos to the servo headers.

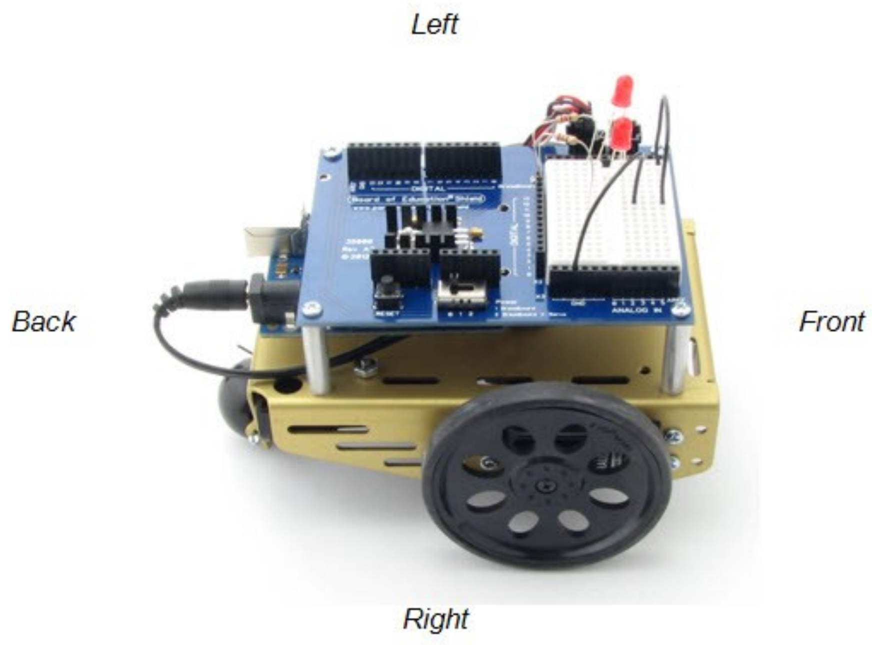




- ✓ From the underside of the chassis, pull any excess servo and battery cable through the rubber grommet hole, and tuck the excess cable lengths between the servos and the chassis.

Re-test the Servos

In this activity, you will test to make sure that the electrical connections between your board and the servos are correct. The picture below shows your BOE Shield-Bot's front, back, left, and right. We need to make sure that the right-side servo turns when it receives pulses from pin 12 and that the left-side servo turns when it receives pulses from pin 13.



Testing the Left and Right Wheels

Testing the Right Wheel

The next example sketch will test the servo connected to the right wheel, shown below. The sketch will make this wheel turn clockwise for three seconds, then stop for one second, then turn counterclockwise for three seconds.

Set the BOE Shield-Bot on its nose so that the drive wheels are suspended above the ground.

Connect the programming cable and battery pack to the Arduino.

Enter, save, and upload RightServoTest to your Arduino

Set the 3-position switch to position-2 and press/release the RESET button.

Verify that the right wheel turns clockwise for three seconds, stops for one second, then turns counterclockwise for three seconds.

If the right wheel/servo does behave properly, then keep going.

```
* Robotics with the BOE Shield - RightServoTest
* Right servo turns clockwise three seconds, stops 1
  second, then  * counterclockwise three seconds.  */

#include <Servo.h>                 // Include servo library

Servo servoRight;                // Declare right servo

void setup()                      // Built in initialization block
{
  servoRight.attach(12);              // Attach right signal to pin 12
  servoRight.writeMicroseconds(1300);         // Right wheel clockwise
delay(3000);                                  // ...for 3 seconds

  servoRight.writeMicroseconds(1500);         // Stay still
delay(1000);                                  // ...for 3 seconds

  servoRight.writeMicroseconds(1700);     // Right wheel counterclockwise
delay(3000);                                  // ...for 3 seconds

  servoRight.writeMicroseconds(1500);     // Right wheel counterclockwise

}

void loop()                                    // Main loop auto-repeats
{                                      // Empty, nothing needs repeating
}
```

Your Turn – Testing the Left Wheel

Now, it's time to run the same tes onthe left wheel as shown below. This involves modifying the RightServoTest sketch.

Save RightServoTest as LeftServoTest.

Change `Servo servoRight` to `Servo servoLeft`.

Change `servoRight.Attach(12)` to `servoLeft.Attach(13)`.

Replace the rest of the `servoRight` references with `servoLeft`.

Save the sketch, and upload it to your Arduino.

Verify that it makes the left servo turn clockwise for 3 seconds, stop for 1 second, then turn counterclockwise for 3 seconds.

Start-Reset Indicator

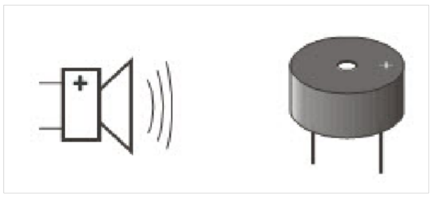
In this activity, we'll build a small noise-making circuit on the BOE Shield's prototyping area that will generate a tone if the robot's batteries run too low.

When the voltage supply drops below the level a device needs to function properly, it's called brownout. The device (your Arduino) typically shuts down until the supply voltage returns to normal. Then, it will restart whatever sketch it was running.

Brownouts typically happen when batteries are already running low, and the servos suddenly demand more power. For example, if the BOE Shield-Bot changes from full speed forward to full speed backward, the servos have to do extra work to stop the servos and then go the other direction. For this, they need more current, and when they try to pull that current from tired batteries, the output voltage dips enough to cause brownout.

Now, imagine your BOE Shield-Bot is navigating through a routine, and suddenly it stops for a moment and then goes in a completely unexpected direction. How will you know if it is a mistake in your code, or if it's a brownout? One simple, effective solution is to add a speaker to your BOE Shield-Bot and make it play a "start" tone at the beginning of every sketch. That way, if your BOE Shield-Bot has a brownout while it's navigating, you'll know right away because it'll play the start tone.

We'll use a device called a piezoelectric speaker (piezospeaker) that can make different tones depending on the
frequency of high/low signals it receives from the Arduino. The schematic symbol and part drawing are shown below.



Frequency is the measurement of how often something occurs in a given amount of time.

A piezoelectric element is a crystal that changes shape slightly when voltage is applied to it. Applying high and low voltages at a rapid rate causes the crystal to rapidly change shape. The resulting vibration in turn vibrates the air around it, and this is what our ear de ects as a tone. Every rate of vibration makes a different tone.

Piezoelectric elements have many uses. When force is applied to a piezoelectric element, it can create voltage. Some piezoelectric elements have a frequency at which they naturally vibrate. These can be used to create voltages at frequencies that function as the clock oscillator for many computers and microcontrollers.

Build the Piezospeaker Circuit

Parts Required

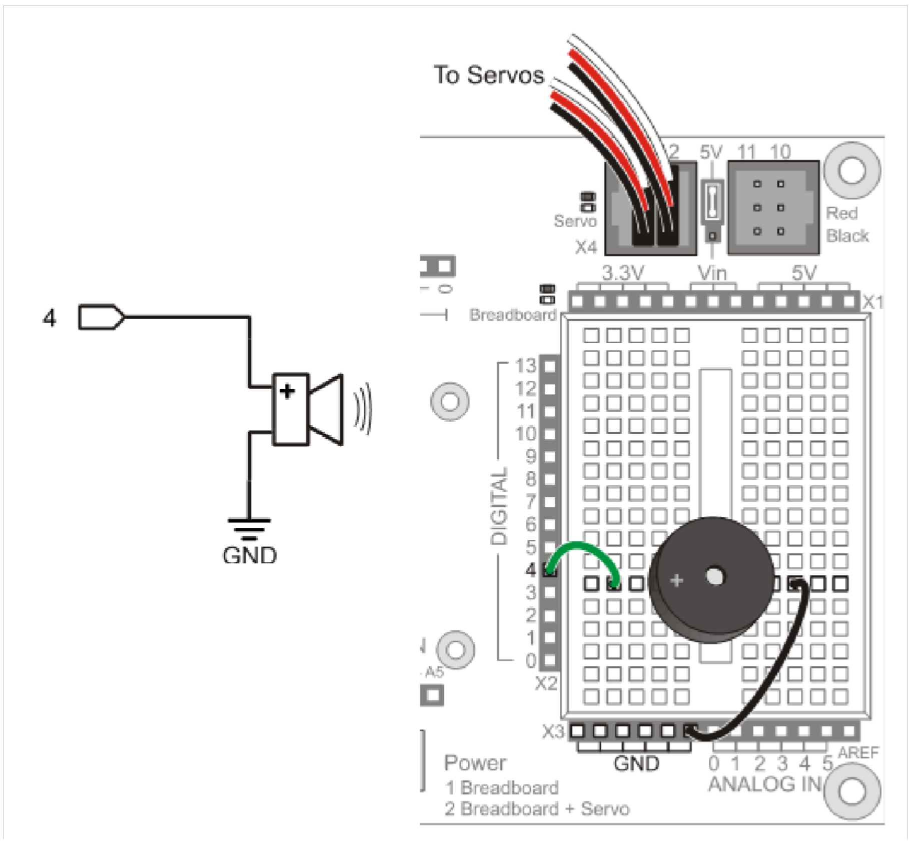
(1) piezospeaker (just peel off the "Remove the seal after washing" sticker
if it has one) (misc.) jumper wires

Building the Start/Reset Indicator Circuit

The picture below shows a wiring diagram for adding a piezospeaker to the breadboard.

Always disconnect power before building or modifying circuits!
Set the Power switch to 0.
Unplug the battery pack.
Unplug the programming cable.

✓ Build the circuit shown below. Position it just as shown on the breadboard. The speaker should stay in place for the rest of the book, while other circuits are added or removed around it.



Programming the Start-Reset Indicator

The next example sketch tests the piezospeaker using calls to the Arduino's `tone` function. True to its name, this function send signals to speakers to make them play tones.

There are two options for calling the `tone` function. One allows you to specify the pin and frequency (pitch) of the tone. The other allows you to specify `pin`, `frequency`, and `duration` (in milliseconds). We'll be using the second option since we don't need the tone to go on indefinitely.

```
tone(pin,frequency)
```

This piezospeaker is
designed to play 4.5 kHz
tones for smoke alarms,
but it can also play a

variety of audible tones and usually sounds best in the 1 kHz to 3.5 kHz range. The start-alert tone we'll use is:

```
tone(4, 3000, 1000);
delay(1000);
```

That will make pin 4 send a series of high/low signals repeating at 3 kHz (3000 times per second). The tone will last for 1000 ms, which is 1 second. The `tone` function continues in the background while the sketch moves on to the next command. We don't want the servos to start moving until the tone is done playing, so the tone command is followed by `delay(1000)` to let the tone finish before the sketch can move on to servo control.

Frequency can be measured in hertz (Hz) which is the number of times a signal repeats itself in one second. The human ear is able to detect frequencies in a range from very low pitch (20 Hz) to very high pitch (20 kHz or 20,000 Hz). One kilohertz is one-thousand-times-per-second, abbreviated 1 kHz.

Example Sketch: StartResetIndicator

This example sketch makes a beep when it starts running, then it goes on to send Serial Monitor messages every halfsecond. These messages will continue indefinitely because they are in the `loop` function. If the power to the Arduino is interrupted, the sketch will start at the beginning again, and you will hear the beep.

- ✓ Reconnect power to your board.
- ✓ Enter, save, and upload StartResetIndicator to the Arduino.
- ✓ If you did not hear a tone, check your wiring and code for errors and try again.
- ✓ If you did hear an audible tone, open the Serial Monitor (this may cause a reset too). Then, push the reset button on the BOE Shield.
- ✓ Verify that, after each reset, the piezospeaker makes a clearly audible tone for one second, and then the "Waiting for reset…" messages resumes.
- ✓ Also try disconnecting and reconnecting your battery supply and programming cable, and then plugging them back in. This should also trigger the start-alert tone.

```
/*
 * Robotics with the BOE Shield -
StartResetIndicator  * Test the piezospeaker
circuit.
 */

void setup()                        // Built in initialization block
{
  Serial.begin(9600);
  Serial.println("Beep!");
tone(4, 3000, 1000);                         // Play tone for 1 second
delay(1000);                                 // Delay to finish tone
}

void loop()                                    // Main loop auto-repeats
{
  Serial.println("Waiting for reset...");
delay(1000);
}
```

How StartResetIndicator Works

StartResetIndicator begins by displaying the message “Beep!” in the Serial Monitor. Then, immediately after printing the message, the `tone` function plays a 3 kHz tone on the piezoelectric speaker for 1 second. Because the instructions are executed so rapidly by the Arduino, it should seem as though the message is displayed at the same instant the piezospeaker starts to play the tone.

When the tone is done, the sketch enters the `loop` function, which displays the same “Waiting for reset…” message over and over again. Each time the reset button on the BOE Shield is pressed or the power is disconnected and reconnected, the sketch starts over again with the “Beep!” message and the 3 kHz tone.

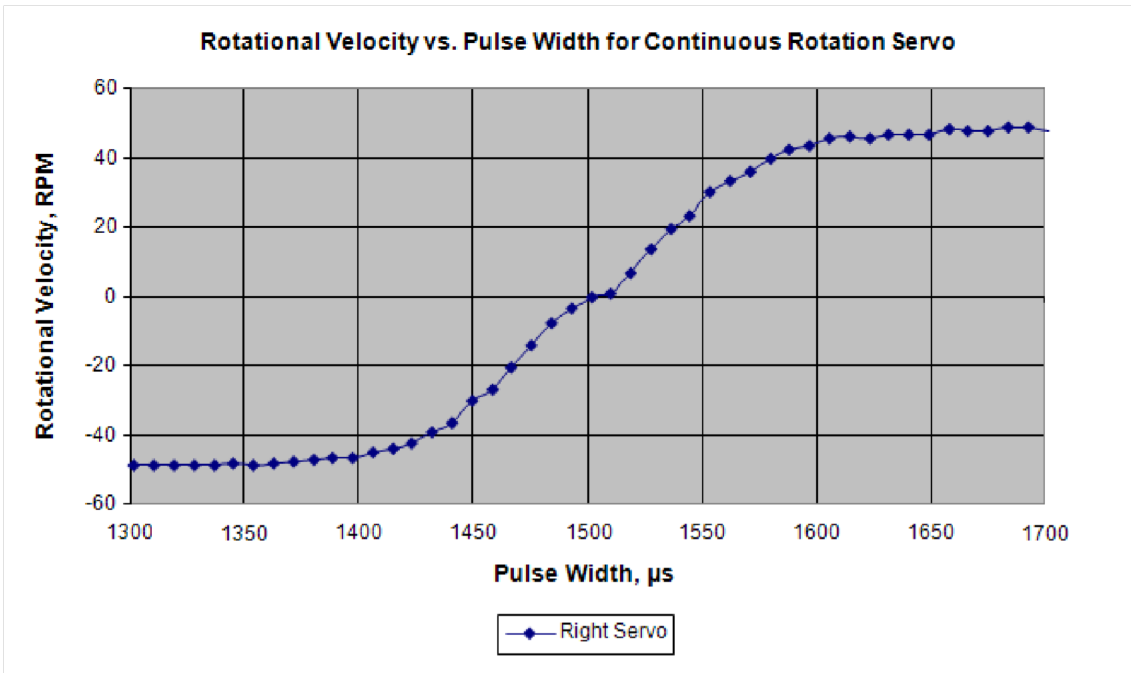
Your Turn – Adding StartResetIndicator to a Different Sketch

We'll use `tone` at the beginning of every example sketch from here onward. So, it’s a good idea to get in the habit of putting `tone` and `delay` statements at the beginning of every sketch’s `setup` function.

- ✓ Open RightServoTest.
- ✓ Copy the `tone` and `delay` function calls from the StartResetIndicator sketch into the beginning of the RightServoTest sketch’s `setup` function.
- ✓ Run the modified sketch and verify that it responds with the “sketch starting” tone every time the Arduino is reset.

Test Speed Control

This graph shows pulse time vs. servo speed. The graph's horizontal axis shows the pulse width in microseconds (µs), and the vertical axis shows the servo's response in RPM. Clockwise rotation is shown as negative, and counterclockwise is positive. This particular servo's graph, which can also be called a transfer curve, ranges from about -48 to +48 RPM over the range of test pulse widths from 1300 to 1700 µs. A transfer curve graph of your servos would be similar.



Three Reasons Why the Transfer Curve Graph is Useful

1. You can get a good idea of what to expect from your servo for a certain pulse width. Follow the vertical line up from 1500 to where the graph crosses it, then follow the horizontal line over and you'll see that there is zero rotation for 1500 µs pulses. We already knew servoLeft.writeMicroseconds(1500) stops a servo, but try some other values.
   -
   - Compare servo speeds for 1300 and 1350 µs pulses. Does it really make any difference?
   What speed would you expect from your servos with 1550 µs pulses? How about 1525 µs pulses?

2. Speed doesn't change much between 1300 and 1400 µs pulses. So, 1300 µs pulses for full speed clockwise is overkill; the same applies to 1600 vs. 1700 µs pulses for counterclockwise rotation. These overkill speed settings are useful because they are more likely to result in closely matched speeds than picking two values in the 1400 to 1600 µs range.

3. Between 1400 and 1600 µs, speed control is more or less linear. In this range, a certain change in pulse width will result in a corresponding change in speed. Use pulses in this range to control your servo speeds.

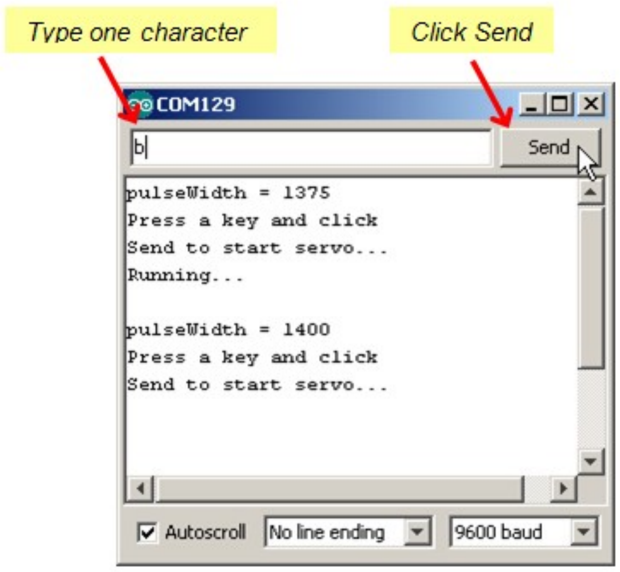
Example Sketch: Test Servo Speed

With this sketch, you can check servo RPM speed (and direction) for pulse values from 1375 µs to 1625 µs in steps of 25 µs. These speed measurements will help make it clear how servo control pulse durations in the 1400 to 1600 µs range control servo speed. This sketch starts by displaying the pulse duration that it's ready to send as a servo control signal. Then, it waits for you to send the Arduino a character with the Serial Monitor before it runs the servo. It runs the servo for six seconds, and during that time you can count the number of full turns the wheel makes. After that, the `for` loop repeats itself, and increases the pulse duration by 25 for the next test.

- ✓ Place a mark (like a piece of masking tape) on the wheel so that you can see how revolutions it turns during the wheel speed tests.
- ✓ Set the BOE Shield-Bot on its nose so that the wheels can spin freely.
- ✓ Enter, save and upload TestServoSpeed to the Arduino.
- ✓ Open the Serial Monitor, and set the drop-down menus to "No line ending"
- ✓ and "9600 baud." Click the transimit pane at the top, type any character, and
- ✓ click the Send button.

Count the number of turns the wheel makes, and multiply by 10 for RPMs. (Don't forget to make a note of direction; it will change after the 5th test.)

- ✓ If you were to add your data points to the graph, would they fit the overall shape?



```
pulseWidth = 1375
Press a key and click
Send to start servo...
Running...

pulseWidth = 1400
Press a key and click
Send to start servo...
```

```
 /*
  Robotics with the BOE Shield - TestServoSpeed
 Send a character from the Serial Monitor to the Arduino to make it
 run the  left servo for 6 seconds.  Starts with 1375 us pulses and
 increases by  25 us with each repetition, up to 1625 us.  This
 sketch is useful for  graphing speed vs. pulse width.  */

 #include <Servo.h>                 // Include servo library

 Servo servoLeft;             // Declare left servo signal

 Servo servoRight;               // Declare right servo signal

 void setup()                            // Built in initialization block
 {
   tone(4, 3000, 1000);                       // Play tone for 1 second
 delay(1000);                               // Delay to finish tone

   Serial.begin(9600);                        // Set data rate to 9600 bps
 servoLeft.attach(13);                      // Attach left signal to P13
 }

 void loop()                                  // Main loop auto-repeats
 {

   // Loop counts with pulseWidth from 1375 to 1625 in increments of 25.

   for(int pulseWidth = 1375; pulseWidth <= 1625; pulseWidth += 25)
   {
     Serial.print("pulseWidth = ");           // Display pulseWidth value
     Serial.println(pulseWidth);
     Serial.println("Press a key and click"); // User prompt
     Serial.println("Send to start servo...");

     while(Serial.available() == 0);          // Wait for character
 Serial.read();                           // Clear character

     Serial.println("Running...");
     servoLeft.writeMicroseconds(pulseWidth); // Pin 13 servo speed=pulse
 delay(6000);                             // ..for 6 seconds
```

```
   servoLeft.writeMicroseconds(1500);       // Pin 13 servo speed = stop

}
}
```

How TestServoSpeed Works

The sketch TestServoSpeed increments the value of a variable named `pulseWidth` by 25 each time through a `for` loop.

```
  // Loop counts with pulseWidth from 1375 to 1625 in increments of 25.

for(int pulseWidth = 1375; pulseWidth <= 1625; pulseWidth += 25)
```

With each repetition of the `for` loop, it displays the value of the next pulse width that it will send to the pin 13 servo, along with a user prompt.

```
    Serial.print("pulseWidth = ");            // Display pulseWidth value
    Serial.println(pulseWidth);
    Serial.println("Press a key and click"); // User prompt
    Serial.println("Send to start servo...");
```

After `Serial.begin` in the `setup` loop, the Arduino sets aside some memory for characters coming in from the Serial
Monitor. This memory is typically called a serial buffer, and that's where ASCII values from the Serial Monitor are stored. Each time you use `Serial.read` to get a character from the buffer, the Arduino subtracts 1 from the number of characters waiting in the buffer.

A call to `Serial.available` will tell you how many characters are in the buffer. This sketch uses `while(Serial.available() = = 0)` to wait until the Serial Monitor sends a character. Before moving on to run the servos, it uses `Serial.read( )` to remove the character from the buffer. The sketch could have used `int myVar = Serial.read( )` to copy the character to a variable. Since the code isn't using the character's value to make decisions, it just calls `Serial.read`, but doesn't copy the result anywhere. The important part is that it needs to clear the buffer so that `Serial.available( )` returns zero next time through the loop.

```
while(Serial.available() == 0);       // Wait for character
Serial.read();                        // Clear character
```

Where is the `while` loop's code block?
The C language allows the `while` loop to use an empty code block, in this case to wait there until it receives a character. When you type a character into the Serial Monitor, `Serial.available` returns 1 instead of 0, so the `while` loop lets the sketch move on to the next statement. `Serial.read` removes that character you typed from the Arduino's serial buffer to make sure that `Serial.available` returns 0 next time through the loop. You could have typed this empty `while` loop other ways:

```
  while(Serial.available() == 0) {}
```

...or:

```
while(Serial.available(
== 0) {};.
```

After the Arduino receives a character from the keyboard, it displays the "Running…" message and then makes the servo turn for 6 seconds. Remember that the `for` loop this code is in starts the `pulseWidth` variable at 1375 and adds 25 to it with each repetition. So, the first time through the loop, `servoLeft` is 1375, the second time through it's 1400, and so on all the way up to 1625.

Each time through the loop, `servoLeft.writeMicroseconds(pulseWidth)` uses the value

that `pulseWidth` stores to set servo speed. That's how it updates the servo's speed each time

you send a character to the Arduino with the Serial Monitor.

```
Serial.println("Running...");
servoLeft.writeMicroseconds(pulseWidth); // Pin 13 speed=pulse
delay(6000);                             // ..for 6 seconds
servoLeft.writeMicroseconds(1500);       // Pin 13 speed=stop
```

Record Your Own Transfer Curve Data

You can use the table below to record the data for your own transfer curve. The TestServoSpeed sketch's loop can be modified to test every value in the table, or every other value to save time.

- ✓ Click the "Printer-friendly version" link at the bottom-right of this page and print out the graph.
- ✓ Change the `for` statement in TestServoSpeed from:

```
            for(int pulseWidth=1375; pulseWidth <= 1625; pulseWidth += 25)
```

…to:

```
    for(int pulseWidth=1300; pulseWidth <= 1700; pulseWidth += 20)
```

- ✓ Load the modified sketch into the Arduino and use it to fill in every other table entry. (If you want to fill in every table entry, **`use pulseWidth += 10`** in the for statement's ***`increment`*** parameter.)
- ✓ Use graphing software of your choice to plot the pulse width vs. wheel RPM.
- ✓ To repeat these measurements for the right wheel, replace all instances of 13 with 12 in the sketch.

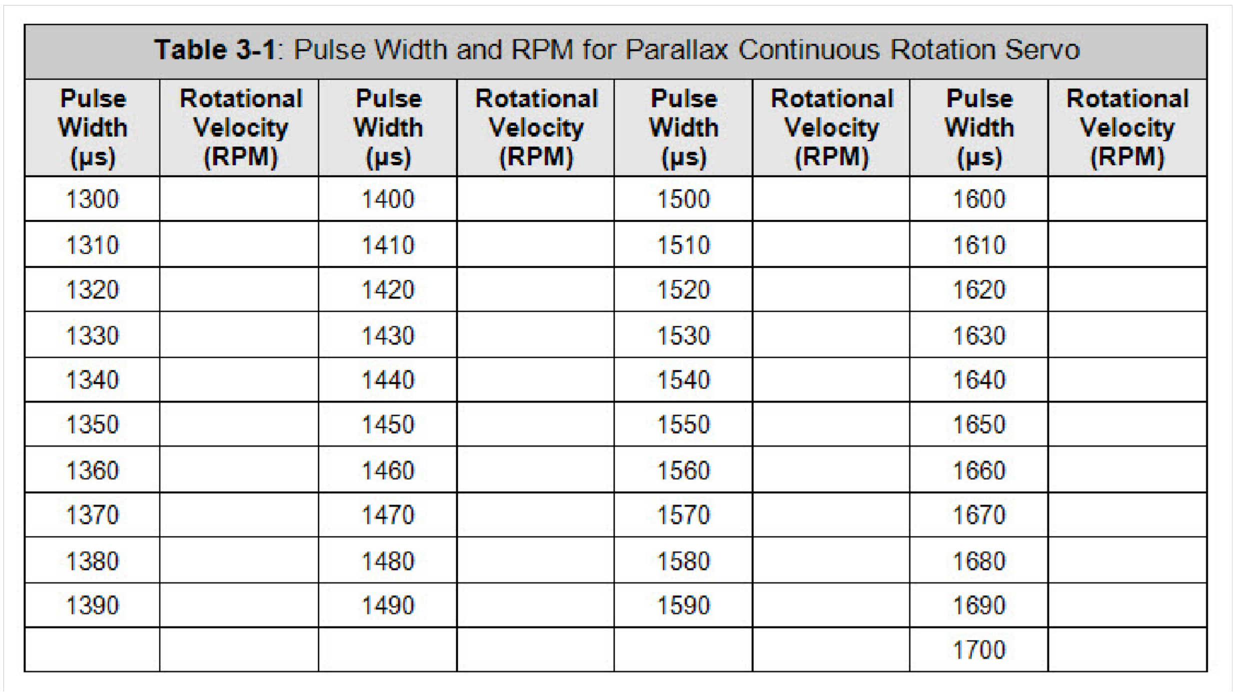


Table 3-1: Pulse Width and RPM for Parallax Continuous Rotation Servo

Pulse Width (µs)	Rotational Velocity (RPM)	Pulse Width (µs)	Rotational Velocity (RPM)	Pulse Width (µs)	Rotational Velocity (RPM)	Pulse Width (µs)	Rotational Velocity (RPM)
1300		1400		1500		1600	
1310		1410		1510		1610	
1320		1420		1520		1620	
1330		1430		1530		1630	
1340		1440		1540		1640	
1350		1450		1550		1650	
1360		1460		1560		1660	
1370		1470		1570		1670	
1380		1480		1580		1680	
1390		1490		1590		1690	
						1700	