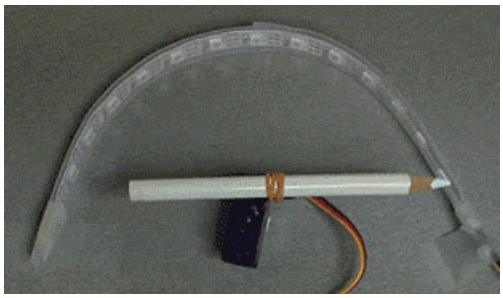
adafruit learning system

Using NeoPixels and Servos Together

Created by Phillip Burgess



Last updated on 2018-08-22 03:44:41 PM UTC

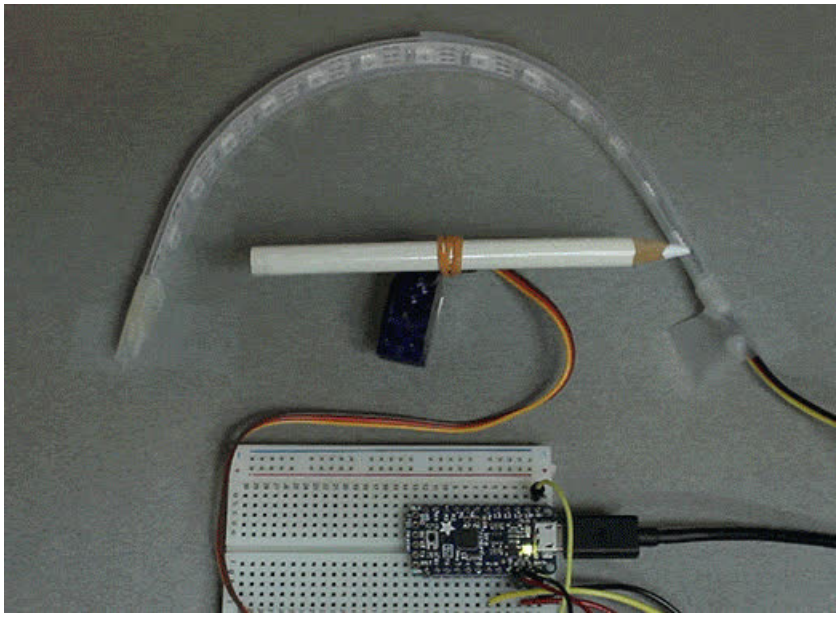
Guide Contents

Guide Contents 2
The Issue 3
The Root of the Problem 3
Using an “M0” Board? 5
Introducing AVR Peripherals 6
Heady Stuff 6
The Good 9
The Bad and The Ugly 9
The TiCoServo Library 11
Special Considerations for Trinket and Gemma 12

The Issue

Not a week goes by in the Adafruit Forums (https://adafru.it/cer) that someone isn't heartbroken to discover that the NeoPixel and Servo libraries won't work together in the same Arduino sketch. Fear not, help is on the way!

Rather than just leave you with a "mysterious black box" workaround (an Arduino library), it also seemed a good opportunity to introduce some advanced Arduino programming concepts. **If you just want a fix and aren't interested in a lengthy technical explanation, that's totally fine! You can skip ahead to the third page, "The TiCoServo Library" and get started.**



The Root of the Problem

The single-wire control protocol used by NeoPixels requires a very steady data stream at 800 kilobits per second. There's a *tiny* margin for error, but not very much. Bits must be issued at a precisely controlled rate...the Adafruit_NeoPixel library handles all this behind the scenes, carefully counting the time of each machine code instruction. For every pixel, there's 24 of *these:*

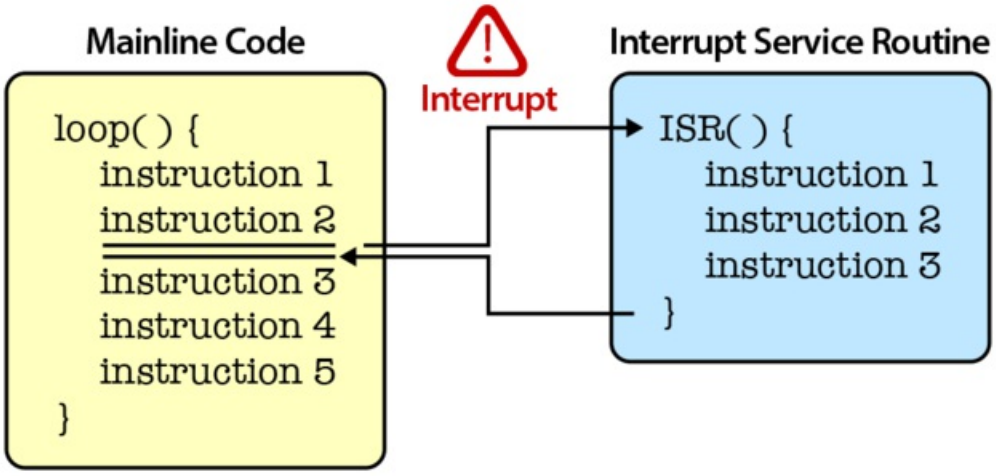
Sequence chart:

Data transfer time(TH+TL=1.25μs±600ns)

T0H	0 code ,high voltage time	0.35us	±150ns
T1H	1 code ,high voltage time	0.7us	±150ns
T0L	0 code , low voltage time	0.8us	±150ns
T1L	1 code ,low voltage time	0.6us	±150ns

Meanwhile, the Arduino is normally using small fractions of time here and there to process *interrupts*...certain events and situations that need to be handled immediately. You usually don't see it, but **interrupts are occurring *all the time* behind the scenes.** Your regular sketch code stops, an *interrupt service routine* is called, and your code then resumes right where it left off. Interrupts help the Arduino's delay() and millis() functions work, as well as Serial.read(), all

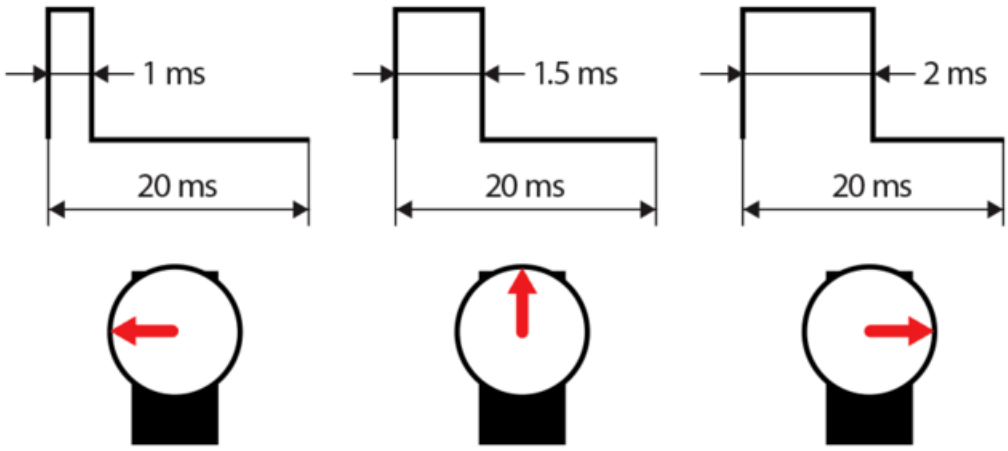
manner of things!



These two concepts cannot coexist. Even a very short and simple interrupt routine will throw off the delicate NeoPixel timing. Therefore, **the NeoPixel library temporarily disables *all* interrupts** when issuing data, then re-enables them when done.

This is rarely a problem. You might have noticed that millis() and micros() lose time in NeoPixel sketches (the timekeeping interrupt stops whenever we write to the strip), that's usually the extent of it.

The issue arises that **servos also have very specific timing requirements** of their own, and the **Arduino Servo library uses interrupts to achieve this**. So every time the NeoPixel library turns off interrupts, even for a moment, the servos will lose track of time (and therefore position) and shake unpredictably. *How sad!*



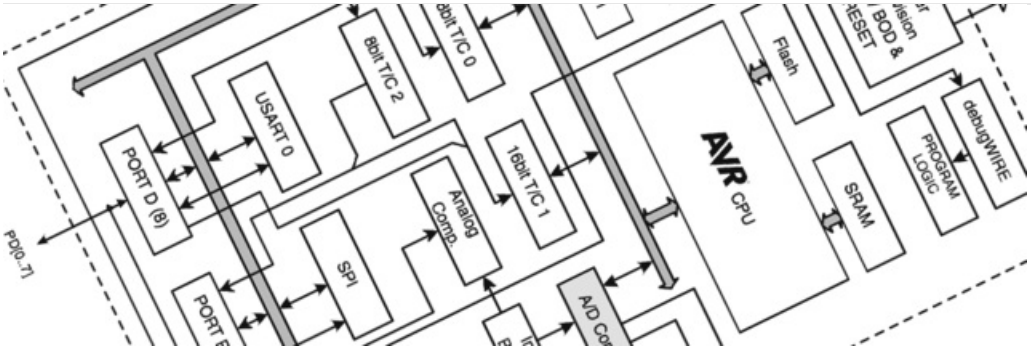
One way to address this is to *use other features of the AVR microcontroller* at the heart of the Arduino to *control the servos without using interrupts,* as we'll explain on the next page. This is an advanced topic, but a good thing to learn about at some point. If the explanation is too technical for your current skill level, or if you'd rather use our library instead, it's okay to skip ahead.

There are hardware-based workarounds as well that are much more flexible. Our 16-channel 12-bit PWM/Servo Driver (in both shield (http://adafru.it/1411) and breakout (http://adafru.it/815) formats) offloads the servo control task to a special-purpose chip, so NeoPixels can't interfere. These boards can "stack" to control dozens (potentially even hundreds) of servos! For complex projects that's probably the way to go. If you just need a couple servos, a little software might be all that's needed...

Using an “M0” Board?

This guide is specific to **8-bit AVR**-based microcontroller boards like the Arduino Uno. If using a newer 32-bit board with an **ARM Cortex-M0** processor (such as the Adafruit Feather M0, Circuit Playground Express or Arduino Zero), we have a different guide using a different approach (https://adafru.it/xBb).

Introducing AVR Peripherals



In normal computer use we usually think of *peripherals* as things like printers or USB drives.

In the realm of the microcontroller, this same word has a slightly different meaning. A “peripheral” is a small section of the silicon real estate, distinct from but alongside the CPU (the part of the microcontroller that actually processes machine instructions), dedicated to handling some specific task *independent* from the CPU.

Some of the peripherals of the Arduino’s AVR microcontroller include an *analog-to-digital converter* (used by the analogRead() function), a *serial UART* (communicates with a host computer, as when using the Serial library and when transferring code to the chip), *SPI* (Serial Peripheral Interface, sometimes used for SD card interfacing among other things) and *I2C* (another chip-to-chip communication method, supported by the Wire library).

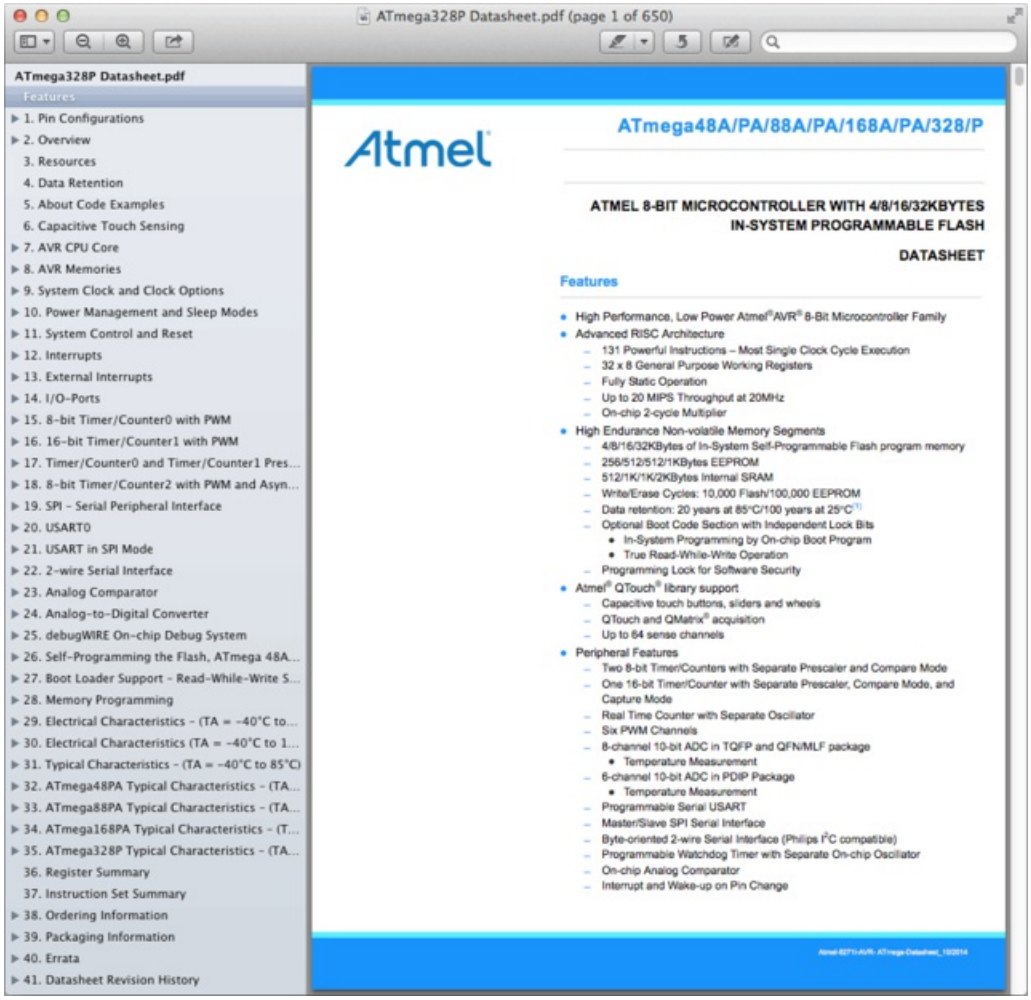
Of interest to us at the moment are the ***Timer/Counter* peripherals**, which precisely measure time intervals and can also be used for pulse-width modulation (PWM, sometimes used for controlling LED brightness or for sound). **PWM output from a timer/counter peripheral can be used to drive a servo without interrupting the CPU.** NeoPixels can coexist! It’s not all roses though…there are some serious limitations…we’ll cover that later.

Heady Stuff

Controlling peripherals directly is very different from normal Arduino programming. For this reason, most are neatly wrapped up in libraries (or the core Arduino library itself, which handles the most oft-used functions like digitalWrite() or analogRead()).

Developing code at this level, one starts with the microcontroller *datasheet*…a massive document detailing every last bit and quantifiable attribute of the chip. Published by (and usually freely downloadable from) the chip manufacturers, these datasheets are unique to every specific chip and variant. For example:

- ATmega 328P Datasheet (https://adafru.it/ecJ) (Arduino Uno, Adafruit Pro Trinket, etc.). 34.3 MB.
- ATmega 32U4 Datasheet (https://adafru.it/ecK) (Arduino Leonardo & Micro, Adafruit FLORA, etc.). 7.5 MB.
- ATmega 2560 Datasheet (https://adafru.it/ecL) (Arduino Mega). 8.4 MB.
- ATtiny85 Datasheet (https://adafru.it/ecM) (Adafruit Trinket & Gemma). 3.8 MB.

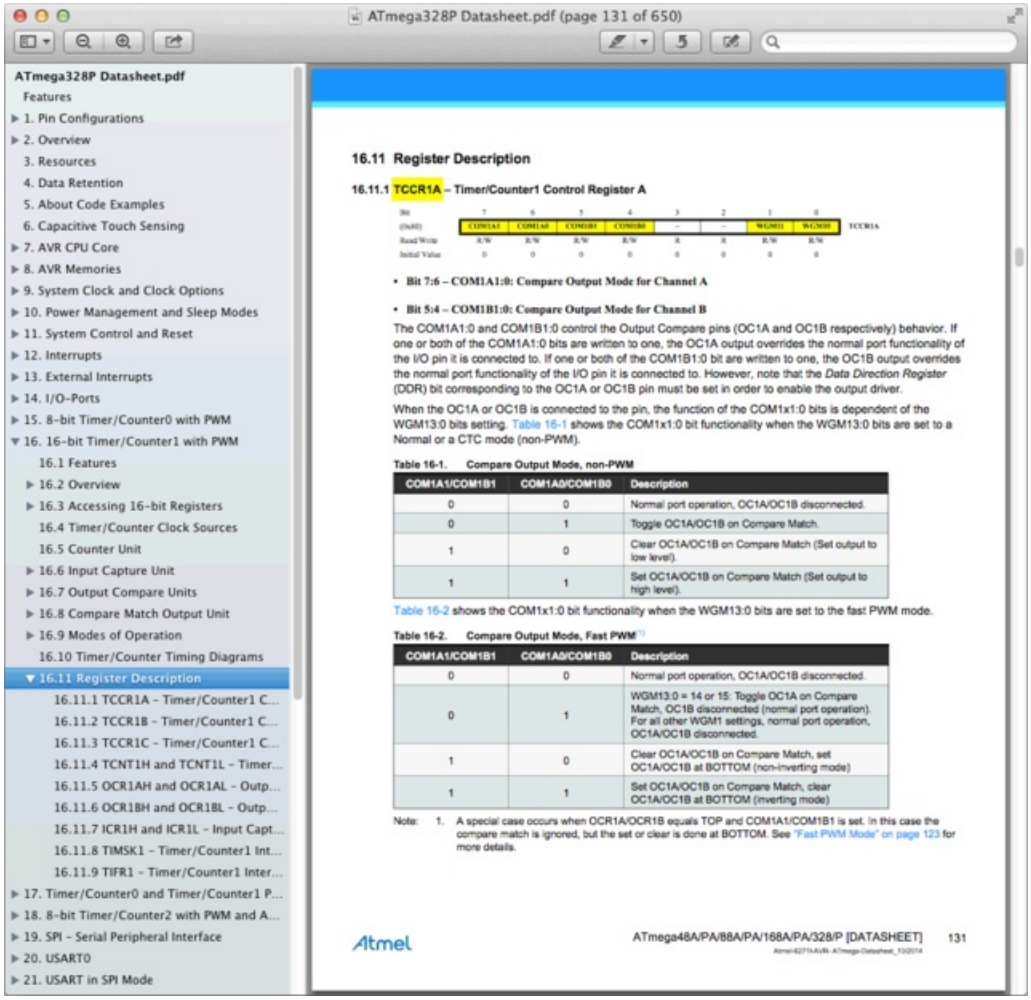


Yes, it really is 650 pages. Fortunately you don't need to read it all. But fascinating to skim!

Peripheral control involves accessing the chip's *special-function registers,* a few dozen memory addresses that can be read, written or modified like variables, but each one…often *individual bits* within each one…control aspects of specific peripherals.

Like variables, the special function registers are referred to by name…these have all been defined in a header file that's automatically included in your sketch.

The ATmega328P in the Arduino Uno and Adafruit Pro Trinket has three timer/counter units (Timer/Counter0, 1 and 2 — each has its own section in the table of contents). Using the 16 MHz CPU clock as a time base, each can count off intervals anywhere between 1 and 256 'ticks' — but Timer/Counter1 is of particular interest because it's a *16-bit counter*…it can count anywhere from 1 to 65,536 ticks, providing lots of extra resolution for this task. The "Register Desription" subsection of the datasheet details each of the special function registers pertaining to Timer/Counter1…the register names and also what each bit controls…



The page shown above describes the special function register named TCCR1A and the individual control bits within. All these control bits have assigned names too…each corresponds to a single bit number…0 to 7…and one must remember when writing C code to use either the _BV(bit) macro or (1 << bit) when identifying bits in the register; multiple bits can be added (+) or OR'd (|) together. Usually several registers need to be configured to get any useful work done.

Here's a few lines from our servo library (which we'll download on the next page) showing what this type of code looks like:

```
TCCR1A = _BV(WGM11);                          // Mode 14 (fast PWM)
TCCR1B = _BV(WGM13) | _BV(WGM12) | _BV(CS11); // 1:8 prescale
ICR1   = F_CPU / 8 / 50;                      // ~50 Hz (~20 ms)
```

Barely even looks like Arduino code, does it? If you've ever had a sketch make you go "buh?", chances are it's directly accessing special function registers.

You'll need to read the relevant parts of the datasheet to fully understand what's going on and why, but basically: the first two lines (setting the special function registers TCCR1A and TCCR1B) configure the the Timer/Counter1's *waveform generation mode* ("fast PWM" in this case) and set a *prescaler* — sort of a throttle on the 'tick' time — advancing the counter every 8 CPU clock cycles instead of every cycle. The next line (ICR1) sets the timer counter's upper limit (after which it restarts at zero) and thus the overall PWM period. The bit of math here factors in the CPU frequency in Hz (F_CPU), prescaler (1:8) and servo pulse frequency (50 Hz) to determine this number…on a 16 MHz Arduino, it would be

16,000,000 ÷ 8 ÷ 50 = 40,000 ticks per PWM cycle.

In other parts of the code are lines like these:

```
TCCR1A ^= _BV(COM1A1);
OCR1A   = pos;
```

The first line toggles (^ is the XOR operator in C) bit COM1A1 in special-function register TCCR1A. This either enables or disables PWM output on the OC1A pin (which is labeled elsewhere in the datasheet…on the Uno, that's pin 9). The second line sets the output compare register for this same pin — its PWM duty cycle — to the value contained in variable "pos."

Obtuse stuff, isn't it? Take it in small steps. Remember, it's all just about setting and clearing bits. Very, very, very *specific* bits. Don't beat yourself up if something doesn't work the first time, or second, or the 23rd…a few projects I've just had to shelve because I never could make heads or tails of it. **AVR peripherals are among the hardest Arduino things to master**, maybe second only to assembly language. This is why Arduino libraries exist, for the rest of us.

Peripherals are a *huge* subject, much more than be covered here (remember, it's a 650 page datasheet), but I wanted to provide a high-level explanation of this very low-level concept.

The Good

The payback for all this hard work? In the case of this library, NeoPixels and servos playing nice together. In the broader sense, *much* more. Byte for byte, cycle for cycle, **there simply is no better optimization strategy than exploiting the microcontroller's built-in peripherals**, period. Once configured and set on their way, *zero code* and *zero instruction* cycles are spent on the task. Other code then runs while the peripherals do their job; it's an explicit form of **multitasking**.

There's a lot more than just PWM and timers…some peripherals offer features not normally exposed by the core Arduino libraries, such as *ADC free-run mode* used in our voice changer project (https://adafru.it/xAl).

The Bad and The Ugly

Performance isn't everything. It oftens comes at a cost…in this case, flexibility. Consider:

- Peripherals and special-function registers are *unique to every single make and model* of microcontroller. To use these is to tie yourself to very specific hardware. Code that performs magic on an Arduino Uno **will not work** on an Arduino Due…**it won't even compile**…they're based on altogether different architectures. Our library works on the most common 8-bit AVR microcontrollers, but cutting-edge "Arduino compatibles" aren't supported (though some of these have entirely different ways of resolving the NeoPixel timing issue).
- Peripherals are an *extremely* limited resource, much more so than even RAM or code space. There is exactly **ONE** 16-bit timer/counter on the Arduino Uno. This can easily lead to **library conflicts**…for example, the WaveHC library (which plays WAV files from an SD card) also relies on Timer/Counter1. It won't get along with NeoPixels either.
- PWM output from the Timer/Counter units is **limited to a very specific set of pins**. On the Arduino Uno, you can control at most **two** servos this way, and they must be on **pins 10 or 11**. On the Leonardo and Micro, at most **four** servos on **pins 5, 9, 10 or 11**. There is no workaround for this that doesn't involve interrupts…putting us back at square one.
- The Trinket and Gemma microcontrollers *don't even have* a 16-bit timer. With only an 8-bit timer to work with, these are limited to about eight distinct servo positions; smooth continuous motion is not possible.

Early versions of the “official” Arduino Servo library worked exactly as we do here...using PWM output from Timer/Counter1. This was later switched to an interrupt-based technique, with the benefit of supporting many servos on any pins. There was no obvious drawback, NeoPixels weren’t even a *thing* until very recently!

The TiCoServo Library

If you just want to download and use the library, that's totally okay. Please just be aware of the following limitations:

- This **only works on certain Arduino-compatible boards**. All the most common hardware with an 8-bit AVR microcontroller should be fine (Arduino Uno, Duemilanove, Leonardo, Mega, Pro Trinket, Teensy 2, etc.). "Cutting edge" boards using other microcontrollers (Arduino Due, Teensy 3, etc.) are right out.
- On **Trinket and Gemma**, only about **eight distinct servo positions** are possible. (Pro Trinket has full resolution.)
- **Servos only work on very specific pins**:

Board	Pins
Arduino Uno, Duemilanove, Diecimila, Adafruit Pro Trinket, Boarduino, Menta (anything w/ATmega328P or ATmega168)	9, 10
Arduino Leonardo, Micro	5, 9, 10, 11
Adafruit FLORA	D9, D10
PJRC Teensy 2.0 (not Teensy+ or 3.X)	4, 9, 14, 15
Arduino Mega	2, 3, 5, 6, 7, 8, 11, 12, 13, 44, 45, 46
Adafruit Trinket	1, 4
Adafruit Gemma	D1

https://adafru.it/e8R

https://adafru.it/e8R

https://adafru.it/cDj

https://adafru.it/cDj

Library installation is a common sticking point for beginners...our All About Arduino Libraries (https://adafru.it/dit) guide explains how this is done.

After installing the libraries, restart the Arduino IDE.

There are two simple examples that mix servos and NeoPixels. One will work on an **Adafruit Gemma or Trinket**, the other on an **Arduino Uno** or most other mainstream boards (Leonardo, etc.). You might need to change some pin numbers (NeoPixel pin #, etc.).

The library is modeled after the official Arduino Servo library...all functions and arguments are identical, and you can simply refer to the Arduino site for reference (https://adafru.it/ecQ). Except for the pin-number limitations, most Arduino Servo sketches should be nearly drop-in compatible with just a few changes:

Instead of:

```
#include <Servo.h>
```

use:

```
#include <Adafruit_TiCoServo.h>
```

The servo declaration changes from:

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

to:

```
Adafruit_TiCoServo myservo; // create servo object to control a servo
```

The attach(), write() and other functions then operate identically to the standard Servo library counterparts...*unless you're using a Trinket or Gemma...*

Special Considerations for Trinket and Gemma

Since they're based on the diminutive ATtiny85 microcontroller, these boards work a little differently.

First, one extra #include line is needed at the top of the code:

```
#include <avr/power.h>
```

Then add the following to the setup() function. It's important that this appear *before* calling servo.attach()!

```
#if (F_CPU == 16000000L)
  clock_prescale_set(clock_div_1);
#endif
```

Unlike the "big boy" code that works with either degrees or microseconds, the "tiny" version can only specify servo positions in raw "ticks," where each tick is equal to about 128 microseconds. Given that most servos nominally want a timing pulse between 1,000 and 2,000 microseconds, that means values from 8 to 15 ticks are a reasonable range. Every servo is a little different though...some have more or less range, so you might be okay adjusting these values slightly.

This isn't the big downer it might seem. Many projects only require two servo positions (e.g. a gate, flag or valve switching between open and closed positions).

With such coarse granularity, degree values are faily meaningless and thus unsupported. You can use the Arduino map() function if you really want to, but whole ranges will just map to a single value. Better to use raw ticks, since you know every value represents a distinct position, even if they don't intuitively relate to a simple degree multiplier.