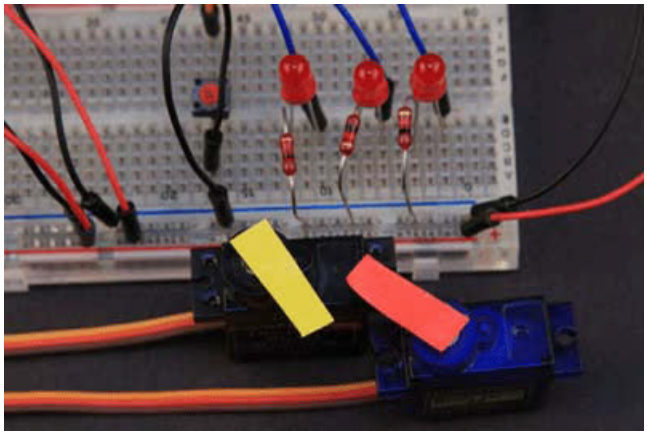
adafruit learning system

Multi-tasking the Arduino - Part 1

Created by Bill Earl



Last updated on 2021-10-22 11:06:46 AM EDT

Guide Contents

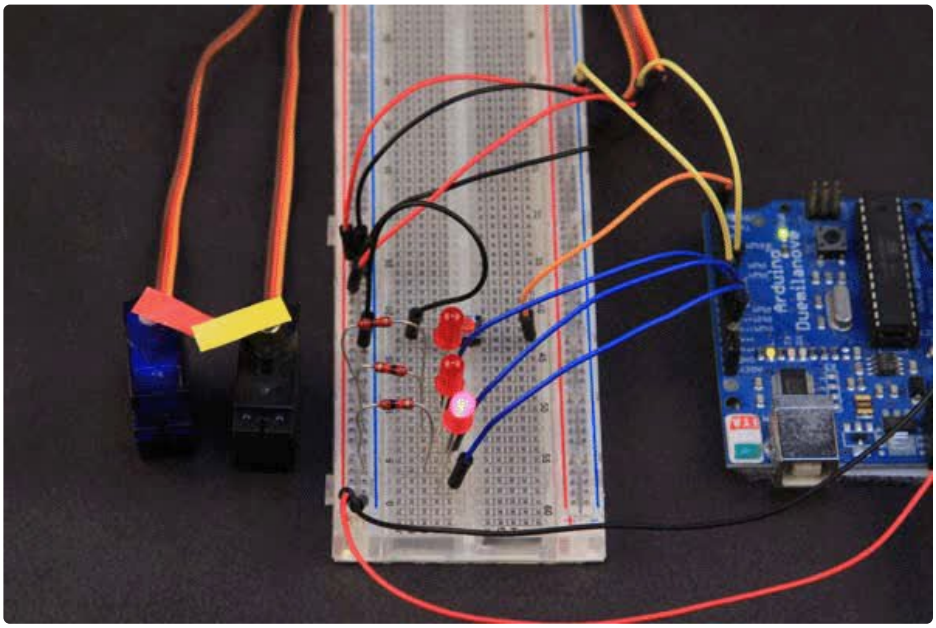
Overview

Bigger and Better Projects

Once you have mastered the basic blinking leds, simple sensors and sweeping servos, it's time to move on to bigger and better projects. That usually involves combining bits and pieces of simpler sketches and trying to make them work together. The first thing you will discover is that some of those sketches that ran perfectly by themselves, just don't play well with others.

The Arduino is a very simple processor with no operating system and can only run one program at a time. Unlike your personal computer or a Raspberry Pi, the Arduino has no way to load and run multiple programs.

That doesn't mean that we can't manage multiple tasks on an Arduino. We just need to use a different approach. Since there is no operating system to help us out, We have to take matters into our own hands.



Ditch the delay()

The first thing you need to do is stop using delay().

Using delay() to control timing is probably one of the very first things you learned when experimenting with the Arduino. Timing with delay() is simple and straightforward, but it does cause problems down the road when you want to add additional functionality. The problem is that delay() is a "busy wait" that monopolizes the processor.

During a delay() call, you can't respond to inputs, you can't process any data and you can't change any outputs. The delay() ties up 100% of the processor. So, if any part of your code uses a delay(), everything else is dead in the water for the duration.

Remember Blink?

```
    /*
  Blink
  Turns on an LED on for one second, then off for one second, repeatedly.

  This example code is in the public domain.
 */

// Pin 13 has an LED connected on most Arduino boards.
// give it a name:
int led = 13;

// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
  pinMode(led, OUTPUT);
}

// the loop routine runs over and over again forever:
void loop() {
  digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);               // wait for a second
  digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);               // wait for a second
}
```

The simple Blink sketch spends almost all of its time in the delay() function. So, the processor can't do anything else while it is blinking.

And sweep too?

Sweep uses the delay() to control the sweep speed. If you try to combine the basic blink sketch with the

servo sweep example, you will find that it alternates between blinking and sweeping. But it won't do both simultaneously.

```
#include <Servo.h>

// Pin 13 has an LED connected on most Arduino boards.
// give it a name:
int led = 13;

Servo myservo;  // create servo object to control a servo
                // twelve servo objects can be created on most boards

int pos = 0;    // variable to store the servo position

void setup()
{
  // initialize the digital pin as an output.
  pinMode(led, OUTPUT);
  myservo.attach(9);  // attaches the servo on pin 9 to the servo object
}

void loop()
{
  digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);               // wait for a second
  digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);               // wait for a second

  for(pos = 0; pos <= 180; pos += 1) // goes from 0 degrees to 180 degrees
  {                                  // in steps of 1 degree
    myservo.write(pos);              // tell servo to go to position in variable 'pos'
    delay(15);                       // waits 15ms for the servo to reach the position
  }
  for(pos = 180; pos>=0; pos-=1)     // goes from 180 degrees to 0 degrees
  {
    myservo.write(pos);              // tell servo to go to position in variable 'pos'
    delay(15);                       // waits 15ms for the servo to reach the position
  }
}
```

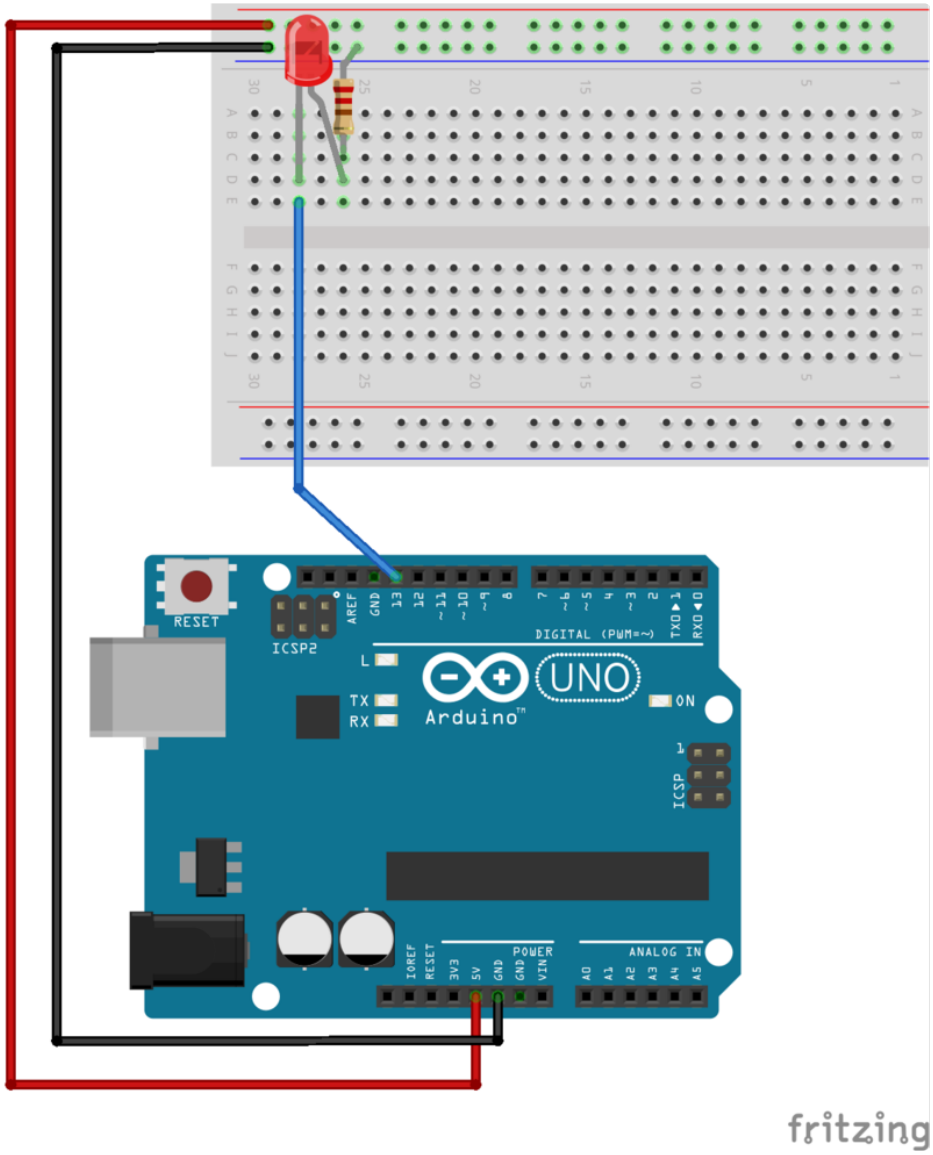
So, how do we control the timing without using the delay function?

Using millis() for timing

Become a clock-watcher!

One simple technique for implementing timing is to make a schedule and keep an eye on the clock. Instead of a world-stopping delay, you just check the clock regularly so you know when it is time to act. Meanwhile the processor is still free for other tasks to do their thing. A very simple example of this is the **BlinkWithoutDelay** example sketch that comes with the IDE.

The code on this page uses the wiring shown in the diagram below:



Blink Without Delay

This is the BlinkWithoutDelay example sketch from the IDE:

```
/* Blink without Delay

 Turns on and off a light emitting diode(LED) connected to a digital
 pin, without using the delay() function.  This means that other code
 can run at the same time without being interrupted by the LED code.

 The circuit:
 * LED attached from pin 13 to ground.
 * Note: on most Arduinos, there is already an LED on the board
 that's attached to pin 13, so no hardware is needed for this example.


 created 2005
 by David A. Mellis
 modified 8 Feb 2010
 by Paul Stoffregen

 This example code is in the public domain.


 http://www.arduino.cc/en/Tutorial/BlinkWithoutDelay
 */

// constants won't change. Used here to
// set pin numbers:
const int ledPin =  13;      // the number of the LED pin

// Variables will change:
int ledState = LOW;             // ledState used to set the LED
long previousMillis = 0;        // will store last time LED was updated

// the follow variables is a long because the time, measured in miliseconds,
// will quickly become a bigger number than can be stored in an int.
long interval = 1000;           // interval at which to blink (milliseconds)

void setup() {
  // set the digital pin as output:
  pinMode(ledPin, OUTPUT);
}

void loop()
{
  // here is where you'd put code that needs to be running all the time.

  // check to see if it's time to blink the LED; that is, if the
  // difference between the current time and last time you blinked
  // the LED is bigger than the interval at which you want to
  // blink the LED.
  unsigned long currentMillis = millis();

  if(currentMillis - previousMillis > interval) {
    // save the last time you blinked the LED
    previousMillis = currentMillis;

    // if the LED is off turn it on and vice-versa:
    if (ledState == LOW)
```

```
      ledState = HIGH;
    else
      ledState = LOW;

    // set the LED with the ledState of the variable:
    digitalWrite(ledPin, ledState);
  }
}
```

What's the point of that?

At first glance, BlinkWithoutDelay does not seem to be a very interesting sketch. It looks like just a more complicated way to blink a LED. However, BinkWithoutDelay illustrates a very important concept known as a **State Machine**.

Instead of relying on delay() to time the blinking. BlinkWithoutDelay remembers the current state of the LED and the last time it changed. On each pass through the loop, it looks at the millis() clock to see if it is time to change the state of the LED again.

Welcome to the Machine

Let’s look at a slightly more interesting blink variant that has a different on-time and off-time. We’ll call this one “FlashWithoutDelay”.

```
// These variables store the flash pattern
// and the current state of the LED

int ledPin =  13;      // the number of the LED pin
int ledState = LOW;             // ledState used to set the LED
unsigned long previousMillis = 0;        // will store last time LED was updated
long OnTime = 250;           // milliseconds of on-time
long OffTime = 750;          // milliseconds of off-time

void setup()
{
  // set the digital pin as output:
  pinMode(ledPin, OUTPUT);
}

void loop()
{
  // check to see if it's time to change the state of the LED
  unsigned long currentMillis = millis();

  if((ledState == HIGH) && (currentMillis - previousMillis >= OnTime))
  {
    ledState = LOW;  // Turn it off
    previousMillis = currentMillis;  // Remember the time
    digitalWrite(ledPin, ledState);  // Update the actual LED
  }
  else if ((ledState == LOW) && (currentMillis - previousMillis >= OffTime))
  {
    ledState = HIGH;  // turn it on
    previousMillis = currentMillis;   // Remember the time
    digitalWrite(ledPin, ledState);   // Update the actual LED
  }
}
```

State + Machine = State Machine

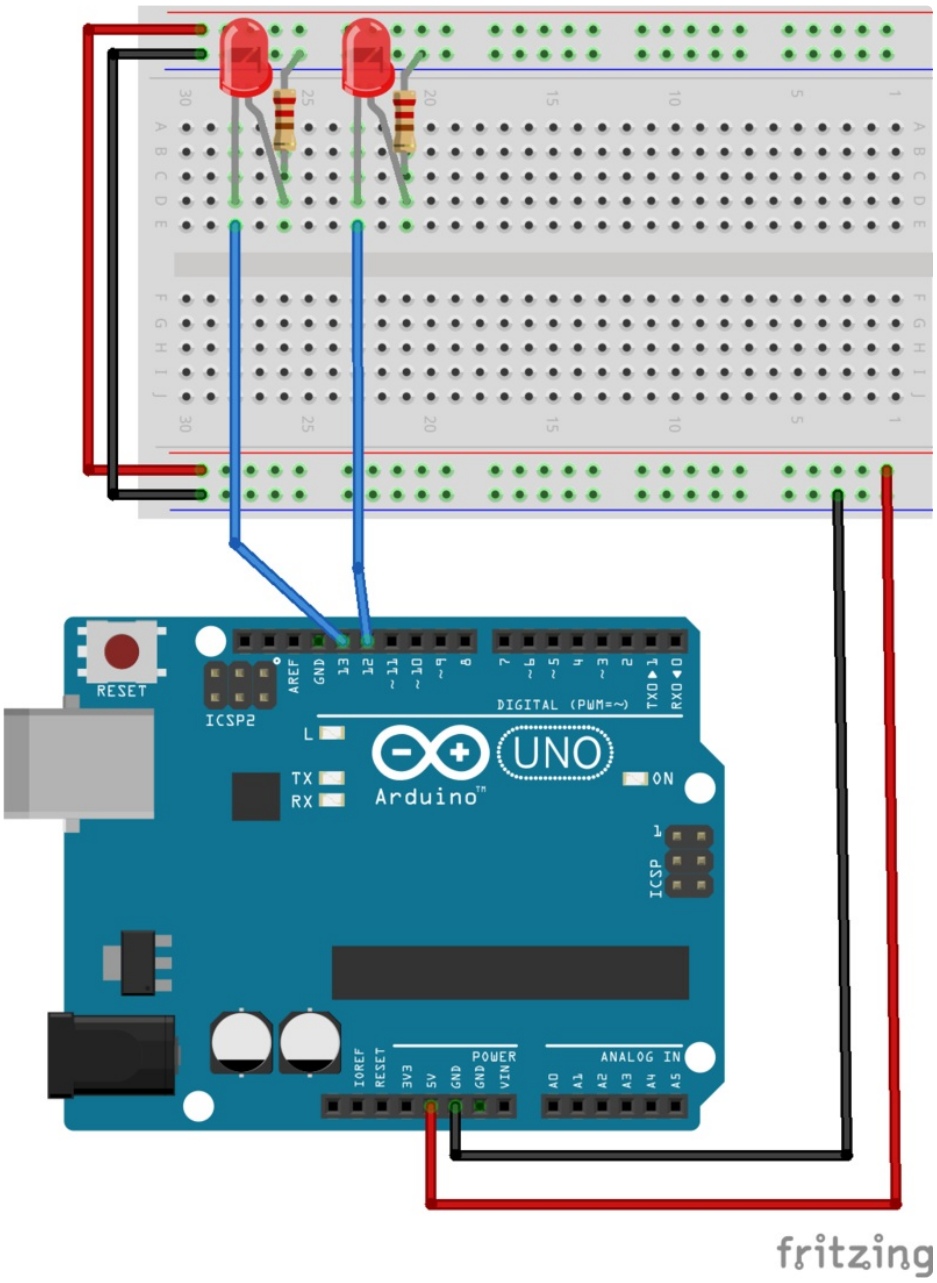
Note that we have variables to keep track of whether the LED is ON or OFF. And variables to keep track of when the last change happened. That is the **State** part of the State Machine.

We also have code that looks at the state and decides when and how it needs to change. That is the **Machine** part. Every time through the loop we ‘run the machine’ and the machine takes care of updating the state.

Next, we'll look at how you can combine multiple state machines and run them concurrently.

Now for two at once

Now it is time to do some multi-tasking! First wire up another LED as in the diagram below.



Then we'll create another state machine for a second LED that flashes at completely different rates. Using two separate state machines allows us to blink the two LEDs completely independent of one another. Something that would be surprisingly complicated to do using delays alone.

```
// These variables store the flash pattern
// and the current state of the LED

int ledPin1 =  12;      // the number of the LED pin
int ledState1 = LOW;             // ledState used to set the LED
unsigned long previousMillis1 = 0;        // will store last time LED was updated
long OnTime1 = 250;           // milliseconds of on-time
long OffTime1 = 750;          // milliseconds of off-time

int ledPin2 =  13;      // the number of the LED pin
int ledState2 = LOW;             // ledState used to set the LED
unsigned long previousMillis2 = 0;        // will store last time LED was updated
long OnTime2 = 330;           // milliseconds of on-time
long OffTime2 = 400;          // milliseconds of off-time

void setup()
{
  // set the digital pin as output:
  pinMode(ledPin1, OUTPUT);
  pinMode(ledPin2, OUTPUT);
}

void loop()
{
  // check to see if it's time to change the state of the LED
  unsigned long currentMillis = millis();

  if((ledState1 == HIGH) && (currentMillis - previousMillis1 >= OnTime1))
  {
    ledState1 = LOW;  // Turn it off
    previousMillis1 = currentMillis;  // Remember the time
    digitalWrite(ledPin1, ledState1);  // Update the actual LED
  }
  else if ((ledState1 == LOW) && (currentMillis - previousMillis1 >= OffTime1))
  {
    ledState1 = HIGH;  // turn it on
    previousMillis1 = currentMillis;   // Remember the time
    digitalWrite(ledPin1, ledState1);   // Update the actual LED
  }

  if((ledState2 == HIGH) && (currentMillis - previousMillis2 >= OnTime2))
  {
    ledState2 = LOW;  // Turn it off
    previousMillis2 = currentMillis;  // Remember the time
    digitalWrite(ledPin2, ledState2);  // Update the actual LED
  }
  else if ((ledState2 == LOW) && (currentMillis - previousMillis2 >= OffTime2))
  {
    ledState2 = HIGH;  // turn it on
    previousMillis2 = currentMillis;   // Remember the time
    digitalWrite(ledPin2, ledState2);   // Update the actual LED
  }
}
```

Thank you sir! May I have another?

You can add more state machines until you run out of memory or GPIO pins. Each state machine can have its own flash rate. As an exercise, edit the code above to add a third state machine.

- First duplicate all of the state variables and code from one state machine.
- Then re-name all the variables to avoid conflicts with the first machine.

It's not very difficult to do. But it seems rather wasteful writing the same code over and over again. There must be a more efficient way to do this!

There are better ways to manage this complexity. There are programming techniques that are both simpler and more efficient. In the next page, we'll introduce some of the more advanced features of the Arduino programming language.

A classy solution

Let's take another look at that last sketch. As you can see, it is very repetitive. The same code is duplicated almost verbatim for each flashing LED. The only thing that changes (slightly) is the variable names.

This code is a prime candidate for a little **O**bject **O**riented **P**rogramming (OOP).

Put a little OOP in your loop.

The Arduino Language is a variant of C++ which supports Object Oriented Programming. Using the OOP features of the language we can gather together all of the state variables and functionality for a blinking LED into a **C++ class**.

This isn't very difficult to do. We already have written all the code for it. We just need to re-package it as a class.

Defining a class:

We start by declaring a "Flasher" class:

Then we add in all the variables from FlashWithoutDelay. Since they are part of the class, they are known as **member variables**.

```
class Flasher
{
 // Class Member Variables
 // These are initialized at startup
 int ledPin;      // the number of the LED pin
 long OnTime;     // milliseconds of on-time
 long OffTime;    // milliseconds of off-time

 // These maintain the current state
 int ledState;               // ledState used to set the LED
 unsigned long previousMillis;   // will store last time LED was updated
};
```

Next we add a **constructor.** The constructor has the same name as the class and its job is to initialize all the variables.

```
class Flasher
{
 // Class Member Variables
 // These are initialized at startup
 int ledPin;      // the number of the LED pin
 long OnTime;     // milliseconds of on-time
 long OffTime;    // milliseconds of off-time

 // These maintain the current state
 int ledState;               // ledState used to set the LED
 unsigned long previousMillis;   // will store last time LED was updated

  // Constructor - creates a Flasher
  // and initializes the member variables and state
  public:
  Flasher(int pin, long on, long off)
  {
 ledPin = pin;
 pinMode(ledPin, OUTPUT);

 OnTime = on;
 OffTime = off;

 ledState = LOW;
 previousMillis = 0;
  }
};
```

Finally we take our loop and turn it into a **member function** called “Update()”. Note that this is identical to our original void loop(). Only the name has changed.

```
class Flasher
{
 // Class Member Variables
 // These are initialized at startup
 int ledPin;      // the number of the LED pin
 long OnTime;     // milliseconds of on-time
 long OffTime;    // milliseconds of off-time

 // These maintain the current state
 int ledState;               // ledState used to set the LED
 unsigned long previousMillis;   // will store last time LED was updated

  // Constructor - creates a Flasher
  // and initializes the member variables and state
  public:
  Flasher(int pin, long on, long off)
  {
 ledPin = pin;
 pinMode(ledPin, OUTPUT);

 OnTime = on;
 OffTime = off;

 ledState = LOW;
 previousMillis = 0;
  }

  void Update()
  {
    // check to see if it's time to change the state of the LED
    unsigned long currentMillis = millis();

    if((ledState == HIGH) && (currentMillis - previousMillis >= OnTime))
    {
     ledState = LOW;  // Turn it off
      previousMillis = currentMillis;  // Remember the time
      digitalWrite(ledPin, ledState);  // Update the actual LED
    }
    else if ((ledState == LOW) && (currentMillis - previousMillis >= OffTime))
    {
      ledState = HIGH;  // turn it on
      previousMillis = currentMillis;   // Remember the time
      digitalWrite(ledPin, ledState);   // Update the actual LED
    }
  }
};
```

By simply re-arranging our existing code into the Flasher class, we have encapsulated all of the variables (the **state**) and the functionality (the **machine**) for flashing a LED.

Now lets use it:

Now, for every LED that we want to flash, we create an **instance** of the Flasher class by calling the

constructor. And on every pass through the loop we just need to call Update() for each instance of Flasher.

There is no need to replicate the entire state machine code anymore. We just need to ask for another instance of the Flasher class!

```
class Flasher
{
 // Class Member Variables
 // These are initialized at startup
 int ledPin;      // the number of the LED pin
 long OnTime;     // milliseconds of on-time
 long OffTime;    // milliseconds of off-time

 // These maintain the current state
 int ledState;               // ledState used to set the LED
 unsigned long previousMillis;   // will store last time LED was updated

  // Constructor - creates a Flasher
  // and initializes the member variables and state
  public:
  Flasher(int pin, long on, long off)
  {
 ledPin = pin;
 pinMode(ledPin, OUTPUT);

 OnTime = on;
 OffTime = off;

 ledState = LOW;
 previousMillis = 0;
  }

  void Update()
  {
    // check to see if it's time to change the state of the LED
    unsigned long currentMillis = millis();

    if((ledState == HIGH) && (currentMillis - previousMillis >= OnTime))
    {
     ledState = LOW;  // Turn it off
      previousMillis = currentMillis;  // Remember the time
      digitalWrite(ledPin, ledState);  // Update the actual LED
    }
    else if ((ledState == LOW) && (currentMillis - previousMillis >= OffTime))
    {
      ledState = HIGH;  // turn it on
      previousMillis = currentMillis;   // Remember the time
      digitalWrite(ledPin, ledState);   // Update the actual LED
    }
  }
};


Flasher led1(12, 100, 400);
```

```
Flasher led2(13, 350, 350);

void setup()
{
}

void loop()
{
 led1.Update();
 led2.Update();
}
```

Less is more!

That's it – ***each additional LED requires just two lines of code!***

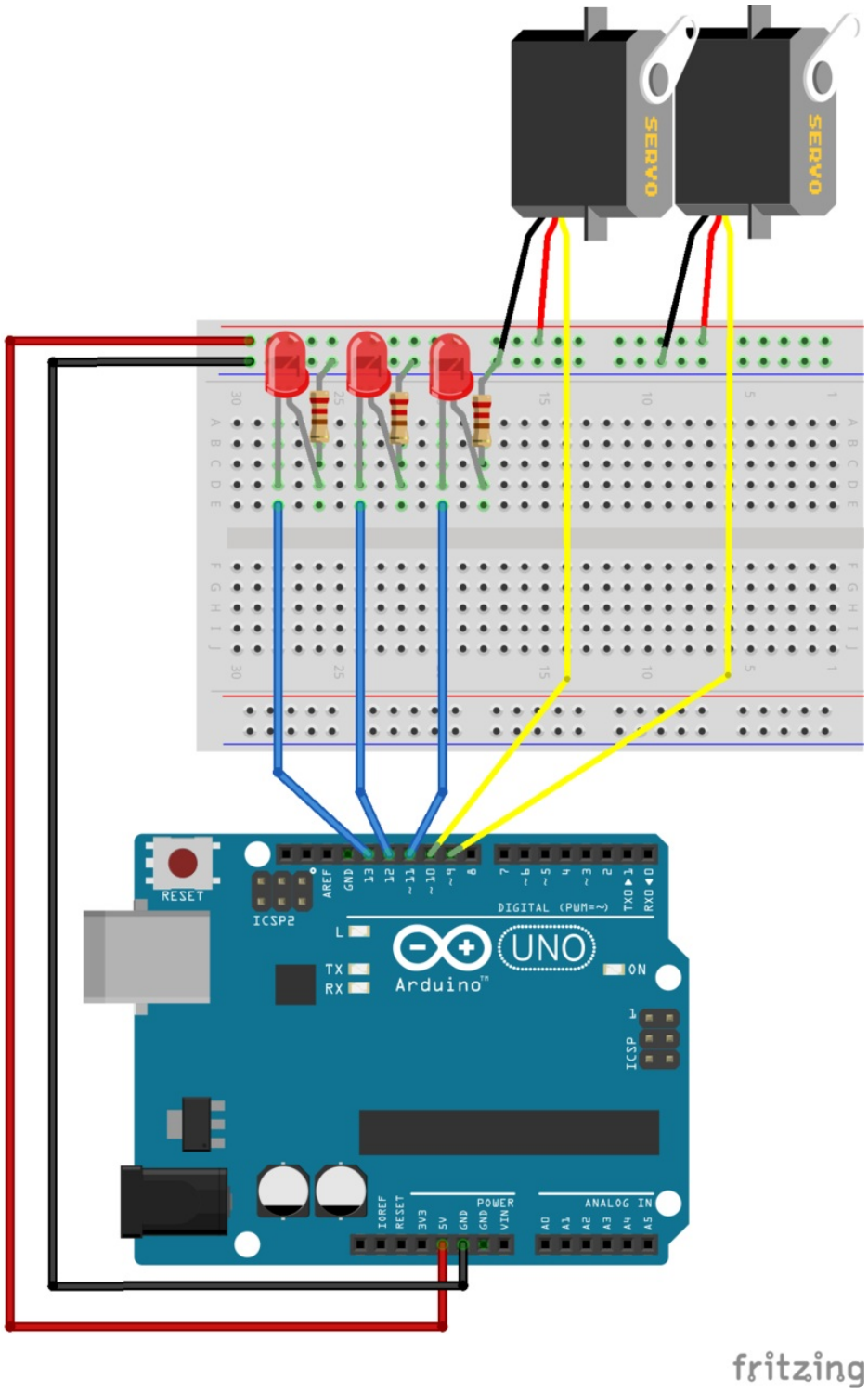
This code shorter and easier to read. And, since there is no duplicated code, ***it also compiles smaller!*** That leaves you even more precious memory to do other things!

A clean sweep

What else can we do with it?

Let’s apply the same principles to some servo code and get some action going.

First hook up a couple of servos on your breadboard as shown below. As long as we are at it, let's hook up a third LED too.



Here is the standard Servo sweep code. Note that it calls the dreaded delay(). We'll take the parts we need from it to make a "Sweeper" state machine.

```
// Sweep
// by BARRAGAN <http://barraganstudio.com>
// This example code is in the public domain.


#include <Servo.h>

Servo myservo;  // create servo object to control a servo
                // a maximum of eight servo objects can be created

int pos = 0;    // variable to store the servo position

void setup()
{
  myservo.attach(9);  // attaches the servo on pin 9 to the servo object
}


void loop()
{
  for(pos = 0; pos < 180; pos += 1)  // goes from 0 degrees to 180 degrees
  {                                  // in steps of 1 degree
    myservo.write(pos);              // tell servo to go to position in variable 'pos'
    delay(15);                       // waits 15ms for the servo to reach the position
  }
  for(pos = 180; pos>=1; pos-=1)     // goes from 180 degrees to 0 degrees
  {
    myservo.write(pos);              // tell servo to go to position in variable 'pos'
    delay(15);                       // waits 15ms for the servo to reach the position
  }
}
```

The Sweeper class below encapsulates the sweep action, but uses the millis() function for timing, much like the Flasher class does for the LEDs.

We also need to add Attach() and Detach() functions to associate the servo with a specific pin:

```
class Sweeper
{
  Servo servo;              // the servo
  int pos;              // current servo position
  int increment;        // increment to move for each interval
  int  updateInterval;      // interval between updates
  unsigned long lastUpdate; // last update of position

public:
  Sweeper(int interval)
  {
    updateInterval = interval;
    increment = 1;
  }

  void Attach(int pin)
  {
    servo.attach(pin);
  }

  void Detach()
  {
    servo.detach();
  }

  void Update()
  {
    if((millis() - lastUpdate) > updateInterval)  // time to update
    {
      lastUpdate = millis();
      pos += increment;
      servo.write(pos);
      Serial.println(pos);
      if ((pos >= 180) || (pos <= 0)) // end of sweep
      {
        // reverse direction
        increment = -increment;
      }
    }
  }
};
```

How many would you like?

Now we can instantiate as many Flashers and Sweepers as we need.

Each instance of a Flasher requires 2 lines of code:

- One to declare the instance
- One to call update in the loop

Each instance of a Sweeper requires just 3 lines of code:

- One to declare the instance
- One to attach it to a pin in setup
- And one call to update in the loop

```
#include <Servo.h>

class Flasher
{
 // Class Member Variables
 // These are initialized at startup
 int ledPin;      // the number of the LED pin
 long OnTime;     // milliseconds of on-time
 long OffTime;    // milliseconds of off-time

 // These maintain the current state
 int ledState;               // ledState used to set the LED
 unsigned long previousMillis;   // will store last time LED was updated

  // Constructor - creates a Flasher
  // and initializes the member variables and state
  public:
  Flasher(int pin, long on, long off)
  {
 ledPin = pin;
 pinMode(ledPin, OUTPUT);

 OnTime = on;
 OffTime = off;

 ledState = LOW;
 previousMillis = 0;
  }

  void Update()
  {
    // check to see if it's time to change the state of the LED
    unsigned long currentMillis = millis();

    if((ledState == HIGH) && (currentMillis - previousMillis >= OnTime))
    {
     ledState = LOW;  // Turn it off
      previousMillis = currentMillis;  // Remember the time
      digitalWrite(ledPin, ledState);  // Update the actual LED
    }
    else if ((ledState == LOW) && (currentMillis - previousMillis >= OffTime))
    {
      ledState = HIGH;  // turn it on
      previousMillis = currentMillis;   // Remember the time
      digitalWrite(ledPin, ledState);   // Update the actual LED
    }
  }
};

class Sweeper
```

```
{
  Servo servo;              // the servo
  int pos;              // current servo position
  int increment;        // increment to move for each interval
  int  updateInterval;      // interval between updates
  unsigned long lastUpdate; // last update of position

public:
  Sweeper(int interval)
  {
    updateInterval = interval;
    increment = 1;
  }

  void Attach(int pin)
  {
    servo.attach(pin);
  }

  void Detach()
  {
    servo.detach();
  }

  void Update()
  {
    if((millis() - lastUpdate) > updateInterval)  // time to update
    {
      lastUpdate = millis();
      pos += increment;
      servo.write(pos);
      Serial.println(pos);
      if ((pos >= 180) || (pos <= 0)) // end of sweep
      {
        // reverse direction
        increment = -increment;
      }
    }
  }
};


Flasher led1(11, 123, 400);
Flasher led2(12, 350, 350);
Flasher led3(13, 200, 222);

Sweeper sweeper1(15);
Sweeper sweeper2(25);

void setup()
{
  Serial.begin(9600);
  sweeper1.Attach(9);
  sweeper2.Attach(10);
}
```

```
void loop()
{
  sweeper1.Update();
  sweeper2.Update();

  led1.Update();
  led2.Update();
  led3.Update();
}
```

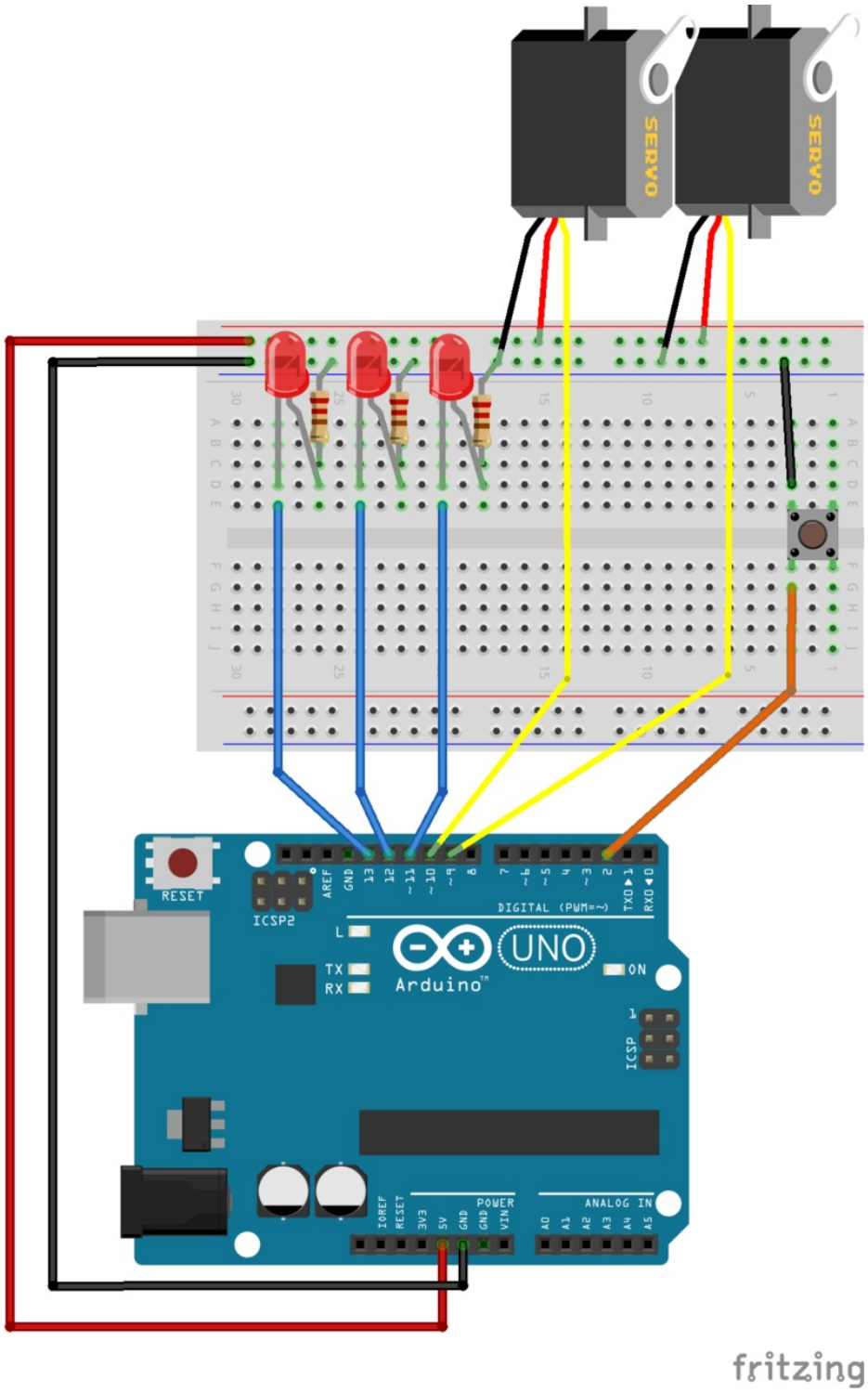
Now we have 5 independent tasks running non-stop with no interference. And our loop() is only 5 lines of code! Next we'll add a button so we can interact with some of these tasks.

All together now!

We want your input too

The other problem with delay()-based timing is that user inputs like button presses tend to get ignored because the processor can't check the button state when it is in a delay(). With our millis()-based timing, the processor is free to check on button states and other inputs regularly. This allows us to build complex programs that do many things at once, but still remain responsive.

We'll demonstrate this by adding a button to our circuit as shown below:



The code below will check the button state on each pass of the loop. Led1 and sweeper2 will not be updated when the button is pressed.

```
#include <Servo.h>


class Flasher
{
```

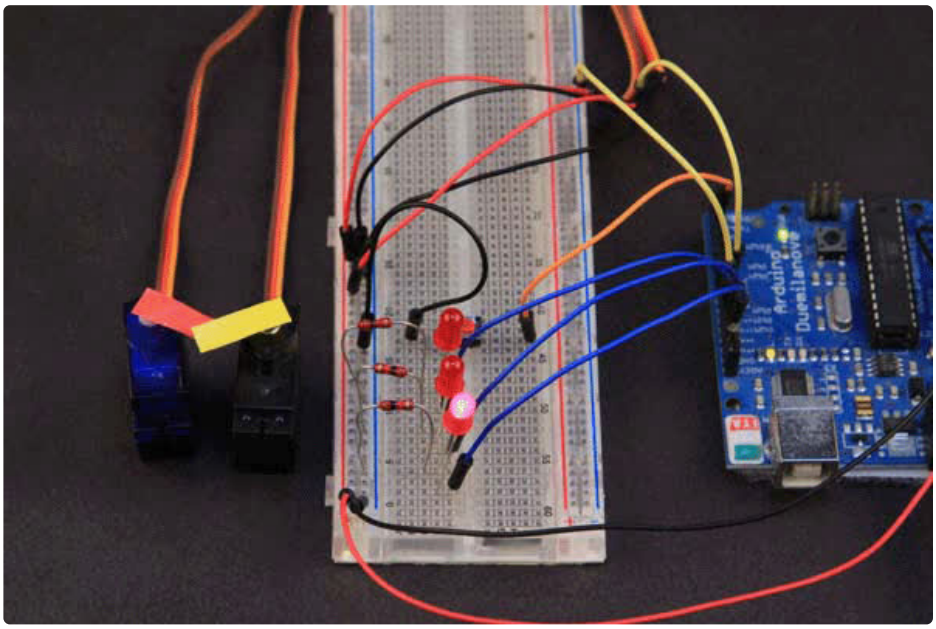
```
{
 // Class Member Variables
 // These are initialized at startup
 int ledPin;      // the number of the LED pin
 long OnTime;     // milliseconds of on-time
 long OffTime;    // milliseconds of off-time

 // These maintain the current state
 int ledState;               // ledState used to set the LED
 unsigned long previousMillis;   // will store last time LED was updated

  // Constructor - creates a Flasher
  // and initializes the member variables and state
  public:
  Flasher(int pin, long on, long off)
  {
 ledPin = pin;
 pinMode(ledPin, OUTPUT);

 OnTime = on;
 OffTime = off;

 ledState = LOW;
 previousMillis = 0;
  }

  void Update()
  {
    // check to see if it's time to change the state of the LED
    unsigned long currentMillis = millis();

    if((ledState == HIGH) && (currentMillis - previousMillis >= OnTime))
    {
     ledState = LOW;  // Turn it off
      previousMillis = currentMillis;  // Remember the time
      digitalWrite(ledPin, ledState);  // Update the actual LED
    }
    else if ((ledState == LOW) && (currentMillis - previousMillis >= OffTime))
    {
      ledState = HIGH;  // turn it on
      previousMillis = currentMillis;   // Remember the time
      digitalWrite(ledPin, ledState);   // Update the actual LED
    }
  }
};

class Sweeper
{
  Servo servo;              // the servo
  int pos;              // current servo position
  int increment;        // increment to move for each interval
  int  updateInterval;      // interval between updates
  unsigned long lastUpdate; // last update of position

public:
  Sweeper(int interval)
  {
```

```
    updateInterval = interval;
    increment = 1;
  }

  void Attach(int pin)
  {
    servo.attach(pin);
  }

  void Detach()
  {
    servo.detach();
  }

  void Update()
  {
    if((millis() - lastUpdate) > updateInterval)  // time to update
    {
      lastUpdate = millis();
      pos += increment;
      servo.write(pos);
      Serial.println(pos);
      if ((pos >= 180) || (pos <= 0)) // end of sweep
      {
        // reverse direction
        increment = -increment;
      }
    }
  }
};


Flasher led1(11, 123, 400);
Flasher led2(12, 350, 350);
Flasher led3(13, 200, 222);

Sweeper sweeper1(15);
Sweeper sweeper2(25);

void setup()
{
  Serial.begin(9600);
  pinMode(2, INPUT_PULLUP);
  sweeper1.Attach(9);
  sweeper2.Attach(10);
}


void loop()
{
  sweeper1.Update();

  if(digitalRead(2) == HIGH)
  {
     sweeper2.Update();
     led1.Update();
  }
```

```
  led2.Update();
  led3.Update();
}
```

The 3 LEDs will flash at their own rates. The 2 sweepers will sweep at their own rates. But when we press the button, sweeper2 and led1 will stop in their tracks until we release the button.

Since there are no delays in the loop, the button input has nearly instantaneous response.



So now we have 5 tasks executing independently and with user input. There are no delays to tie up the processor. And our efficient Object Oriented code leaves plenty of room for expansion!

Conclusion:

In this guide we have demonstrated that it is indeed possible for the Arduino to juggle multiple independent tasks while remaining responsive to external events like user input.

- We've learned how to time things using millis() instead of delay() so we can free up the processor to do other things.
- We've learned how to define tasks as state machines that can execute independently of other state machines at the same time.
- And we've learned how to encapsulate these state machines into C++ classes to keep our code simple and compact.

These techniques won't turn your Arduino into a supercomputer. But they will help you to get the most out of this small, but surprisingly powerful little processor.

In the part 2 of this series, we'll build on these techniques and explore other ways to make your Arduino responsive to external events while managing multiple tasks.