

1 Goal: A Breathing LED Indicator

On December 2, 2003, Apple was awarded US Patent number 6,658,577 for an *Breathing status LED indicator*. When the lid of a laptop was closed, the pulsing LED indicated that the laptop was in a suspended state. Other manufacturers copied this pattern with blinking indicator lights, while Apple used the distinctive breathing LED pattern. Current models of the Apple laptops do not use the pulsing/breathing LED indicator.

The goal of this exercise is to develop an Arduino program to imitate the *old* Apple LED indicator. By completing this exercise you will learn how to use mathematical formulas to control an Arduino output signal, and to use the internal system clock to provide a time base for evaluating those formulas.

1.1 Learning Objectives

These notes present a series of Arduino programming snippets and complete sketches that implement aspects of the breathing LED. The final steps in this project are left to you to complete. The Arduino codes presented in this document presume that you already understand

- Basic Arduino program structure;
- Syntax of `if` constructs;
- How to build a circuit for blinking an LED with Arduino;
- Use of PWM to control an LED.

As a result of following the examples in this document, and practicing with the codes provided here, you should be able to

1. Explain why using the `delay` function will not help with the time-based control of the LED brightness;
2. Read the system clock with `millis` and `micros`;
3. Use the remainder operator (`%`) and the system clock, which increases continuously, to find the time within a repeating cycle;
4. Use the `%` operator and the system clock to achieve a repeating pattern of three constant LED brightness levels;
5. Describe and use formulas for linear and exponential variations during the inhale and exhale phases of a cyclically repeating pattern brightness levels.

Following the examples in this document will give you a series of codes of increasing complexity. We strongly recommend that you save each code as a separate sketch rather than continuously modifying the same sketch. By saving separate codes for each example, you will be able to revisit these notes and review your own intermediate steps. You will also have working code to revert to if, as you add new features, you find yourself with a severely broken code. In that case you can discard broken code and start over from an earlier, saved version.

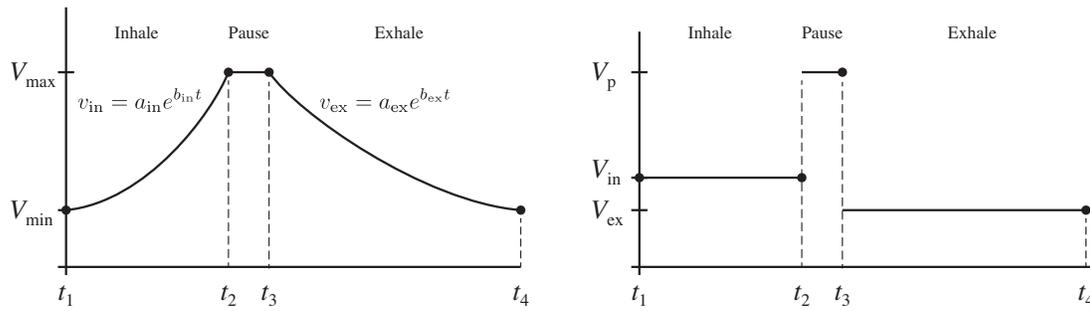


Figure 1: Mathematical representation of breathing (left) and simplified three-level model (right) used to develop code logic.

1.2 Timing and Light Intensity Patterns of the Breathing LED

The diagram in the left side of Figure 1 shows the desired pattern of light intensity to simulate a breathing LED. The breathing pattern has three phases: inhale, pause, and exhale. The inhale and exhale phases are modeled with exponentially increasing and exponentially decreasing functions of time. The pattern repeats every cycle that is t_4 units of time. The inhale phase ends at t_1 , and the exhale phase begins at t_3 .

We approach the code development in two steps. First, we create the code necessary to split the repeating cycle of the breathing into three phases that are controlled by reading the system clock on the Arduino. For that purpose, we use the simplified diagram on the right side of Figure 1, where each breathing phase is indicated by a constant LED brightness. Once the timing logic is created, in the step of code development we add mathematical functions to control the brightness of the LED within the inhale and exhale phases. The final code is rather simple, but we think that it is good practice to develop code in an orderly, stepwise process.

1.3 The LED Circuit

This project uses a basic LED circuit as depicted in the schematic in Figure 2. To control the brightness of the LED, one of the PWM output pins (digital outputs 3, 5, 6, 9, 10 or 11 on an Arduino UNO) is connected to the LED, which is tied to ground by a current-limiting resistor. A $330\ \Omega$ resistor is recommended for an UNO, which supplies 5V for a digital HIGH signal.

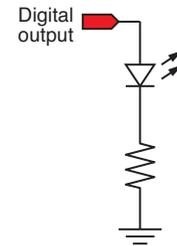


Figure 2: Circuit for control of an LED with digital I/O.

2 Constant Brightness Levels

To establish a starting code structure, and to verify that the electrical circuit is working correctly, we first develop a very simple code that uses the `delay` function to control the duration of a constant brightness level. As depicted in right half of Figure 1, the light intensity of the LED changes from V_{in} to V_p and then to V_{ex} during each cycle. The time durations for the three phases is arbitrary, and we strongly suggest that you use variables (as opposed to numerical constants) to control both the light intensity and the timing.

The value of V can be thought of as a voltage. However, for convenience we never convert the V values to voltage. Instead, we use the 8-bit scale ($0 \leq V \leq 255$) of the PWM output channel.

The values of t are naturally expressed in seconds. In the code, time is measured in milliseconds. Therefore, you will need to translate between seconds and milliseconds, or work only in milliseconds.

2.1 Basic Timing with the delay Function

To implement the three step brightness pattern, the code to the right *could* be used in the `loop` function of an Arduino sketch. The three brightness levels mark the three phases of the breath cycle, and are only meant to provide visual indication of how the code is (or is not) working. The PWM outputs for those three phases have the arbitrary numerical values `Vin`, `Vpause`, and `Vex`. The corresponding time durations in milliseconds are `Tin`, `Tpause` and `Tex`.

```
int Vin = 120, Vpause = 250, Vex = 20;
int Tin = 2000, Tpause = 1500, Tex = 3000;

analogWrite(LED_pin, Vin);
delay(Tin);

analogWrite(LED_pin, Vpause);
delay(Tpause);

analogWrite(LED_pin, Vex);
delay(Tex);
```

The values of `Vin`, `Vpause`, and `Vex` must be in the range $0 \leq V \leq 255$ because the second argument of the `analogWrite` function is an 8-bit value. The argument of the `delay` function is the time to pause in milliseconds.

You could use this code to experiment with the values of `Vin`, `Vpause`, and `Vex` to develop a feel for the brightness levels obtained with different values of the PWM duty cycle. The perceived brightness is not linearly related to the value of the duty cycle.

2.2 Reading the System Clock with millis and micros

The `delay` function has a key disadvantage: the Arduino program can do nothing else while the delay timer is running down. In other words, `delay` *blocks* the Arduino for doing any useful work until the time specified in the `delay` function has passed. For the simple `blink` sketch, blocking execution is not a problem. However, for more complex control problems the `delay` function is not helpful.

We will present an alternative approach that uses the system clock to time the three breathing intervals. The key advantage of using the on-board clock is that it makes timing of the breathing computations independent of any other operations you may wish to have the Arduino perform.

Arduino boards have an on-board system clock that counts the time since the Arduino was supplied with power, or since the reset button was pushed. The system clock will be used to determine which part of the breathing cycle to display by adjusting the LED brightness. The system clock can be read with two functions, `millis()` and `micros()`.

- `millis()` Returns an integer equal to the number of milliseconds that have elapsed since the last system reset
- `micros()` Returns an integer equal to the number of microseconds that have elapsed since the last system reset

The values returned by `millis` or `micros` should be assigned to `unsigned long` variables because the system clock values can be large numbers. Table 1 shows the largest times that can be stored by `unsigned int` and `unsigned long` values. An `unsigned int` can store time values up to a little over one minute. An `unsigned long` can store time values up to almost 50 days. The situation is more extreme when the `micros` function is used. For the breathing LED project, the `millis` function is appropriate since the overall time intervals are measured in seconds.

Table 1: Maximum times that can be stored in Arduino integer variables when time in milliseconds is returned by `millis`.

	Maximum value	Maximum Time			
		Seconds	Minutes	Hours	Days
<code>unsigned int</code>	65536	65.536	1.09	0.018	$\ll 1$
<code>unsigned long</code>	4294967296	4924967	71583	1193	49.7

Listing 1 shows a simple Arduino sketch to read the system clock and print values from those readings to the Serial Monitor. If this code runs for more than one minute, the values `Inow` in the first column attain a maximum of 65536 and then wrap around to zero again. As an exercise, compute the time (that you would observe on your watch) that must elapse before the value of `Mnow` would wrap around to zero¹.

¹*Hint* (1): What is the largest value that can be stored in an `unsigned int`?

Hint (2): The answer is a little more than one hour.

```
// File: readSystemClock.ino
//
// Demonstrate reading of the system clock and storing the results
// in different types of integer variables

void setup() {
  Serial.begin(9600);
}

void loop() {

  unsigned int Inow;
  unsigned long Unow, Mnow;

  // -- Read system clock
  Inow = millis();      // millis() returns the clock in milliseconds
  Unow = millis();
  Mnow = micros();     // micros() returns the clock reading in microseconds

  // -- Print values stored in the different integer types
  Serial.print(Inow);
  Serial.print(" ");
  Serial.print(Unow);
  Serial.print(" ");
  Serial.println(Mnow);

  delay(100);          // Use delay to slow down screen output
}
```

Listing 1: Arduino sketch that demonstrates how to read the system clock.

```
// Function: demoRem.ino
//
// Demonstrate use of the remainder (or modulo) operator

void setup() {

  int num, denom=5, rem;

  Serial.begin(9600);

  for ( num = 1; num<=9; num++ ) {
    rem = num % denom;
    Serial.print(num);
    Serial.print(" "); Serial.print(denom);
    Serial.print(" "); Serial.println(rem);
  }
}

void loop() { } // Empty on purpose.
```

Listing 2: Simple Arduino sketch that demonstrates the remainder operator.

2.3 Confine Time Values to a Single Breath Cycle

To simulate breathing, the pattern of changing LED brightness needs to repeat over the cycle of one simulated breath. The system clock as read by `millis` increases indefinitely. We can use the system clock if we use the remainder operator, `%` to truncate the reading of the system clock to the time interval of one breath cycle.

The remainder operator takes two integer values and returns the remainder of dividing the first number by the second. Let `num` be the numerator and `denom` be the denominator. The following statement computes the remainder of dividing `num` by `denom`

```
rem = num % denom;
```

The `demoRem` code in Listing 2 is an example of using the remainder function. The value of `denom` is held constant at 5 while `num` is incremented by 1. Running `demoRem` produces the following output in the Serial Monitor.

```
1 5 1
2 5 2
3 5 3
4 5 4
5 5 0
6 5 1
7 5 2
8 5 3
9 5 4
```

The third column is the value of `rem` which is never greater than `denom`. As `num` increases, the remainder increases until `num` can be divided by `denom` without a remainder.

The remainder operator makes it easy to use the system clock to compute the time in a repeating cycle. The `demoCycleTime` sketch in Listing 3 shows how that can be done. The `while` loop allows the calculations to continue until the system clock reads 12500 milliseconds (12.5 seconds). This stopping value is arbitrary and is chosen only to keep the display in the Serial Monitor from scrolling off the screen. To demarcate the boundaries of the while loop, `Start!` and `Stop!` messages are written to the Serial Monitor. Running `demoCycleTime` produces the following output

```

// Function: demoCycleTime.ino
//
// Use the system clock and the remainder operator to compute
// the time in cyclically repeating pattern

void setup() {

  unsigned long now=0, Tcycle=5000, t;

  Serial.begin(9600);
  Serial.println("Start!");

  while ( now < 12500 ) {      // Repeat calculations for 12.5 seconds
    now = millis();           // Read the system clock
    t = now % Tcycle;         // t is the time limited to Tcycle
    Serial.print(now);
    Serial.print(" "); Serial.print(Tcycle);
    Serial.print(" "); Serial.println(t);
    delay(750);
  }

  Serial.println("Done!");
}

void loop() { } // Empty on purpose.

```

Listing 3: Simple Arduino sketch that demonstrates the remainder operator.

```

Start!
0 5000 0
749 5000 749
1500 5000 1500
2250 5000 2250
3002 5000 3002
3752 5000 3752
4503 5000 4503
5254 5000 254
6004 5000 1004
6755 5000 1755
7505 5000 2505
8256 5000 3256
9007 5000 4007
9757 5000 4757
10508 5000 508
11258 5000 1258
12009 5000 2009
12760 5000 2760
Done!

```

The first column is the value of the system clock stored in `now`, the second column is the constant value, `Tcycle`, and the third column is the value of `t`,

```
t = now % Tcycle;
```

An arbitrary delay of 750 milliseconds is introduced to slow the display so that the printed values do not scroll off the screen. Although the value of `now` increases indefinitely², the third column (the value of `t` never exceeds the cycle time, `tCycle`).

²In fact, the value of `millis()` will wrap around after 4294967296 milliseconds.

2.4 Constant LED levels *without* using the delay Function

The `demoNoDelay` sketch in Listing 4 builds on the code from previous sections. The switching of LED brightness levels comes from the code snippet in Section 2.1. The timing relative to a repeating cycle is from the `demoCycleTime` sketch in Listing 3.

Note that the `demoRem` sketch in Listing 2 and the `demoCycleTime` sketch in Listing 3 kept the timing code in the `startup` function so that the demonstration code would only be executed once. That makes viewing the results of the demonstration easier. For the breathing LED display, we need the pattern of LED brightness changes to repeat indefinitely. Therefore, the code for cycle timing in the `demoNoDelay` sketch is in the `loop` function, which is executed repeatedly.

On each pass through the `loop` function in `demoNoDelay`, the system clock is read, and the value of `t` is computed with

```
t = now % Tcycle;
```

Thus, `t` is confined to $0 \leq t < T_{\text{cycle}}$. The compound `if...else if...else` structure compares the value of `t` to the boundaries of the time interval that define the inhale and pause segments of the breathing cycle. The `else` clause takes care of the interval $T_{\text{in}} + T_{\text{pause}} < t < T_{\text{cycle}}$.

```
// File: demoNoDelay.ino
//
// Use PWM to control the brightness of an LED in a cyclically
// repeating pattern of three brightness levels. Timing is controlled
// by reading the system clock and without using the delay() function.

int LED_pin = 11; // Use pin 3, 5, 6, 9, 10 or 11 for PWM

void setup() {
  pinMode(LED_pin, OUTPUT); // Initialize pin for output
}

void loop() {

  int Vin=120, Vpause=250, Vex=20; // 8-bit output values for PWM duty cycle
  int Tin=2000, Tpause=1500, Tex=3000; // Time intervals in milliseconds
  int t, Tcycle;
  unsigned long now; // max unsigned long is a big number

  Tcycle = Tin + Tpause + Tex; // Total length of one cycle

  now = millis(); // Read the system clock
  t = now % Tcycle; // Restrict time to interval of one cycle

  if ( t<Tin ) {
    analogWrite(LED_pin, Vin); // Inhale brightness
  } else if ( t < (Tin+Tpause) ) {
    analogWrite(LED_pin, Vpause); // Pause brightness
  } else {
    analogWrite(LED_pin, Vex); // Exhale brightness
  }
}
```

Listing 4: Arduino sketch that uses the system clock to time a cyclically repeating pattern of LED brightness.

3 Variable Brightness During Inhale and Exhale Phases.

The concepts and code in Section 2.4 introduce the logic to implement the three different breathing phases. The `demoNoDelay` function uses distinct, but constant LED brightness levels for each phase. The next step in the code development is to introduce brightness levels that vary with time to represent the inhale and exhale phases.

3.1 Linear Brightness Variation During Inhale and Exhale

We start by introducing a linear variation of brightness with time for the inhale and exhale phases. Although the linear variation in LED brightness is not quite as natural in appearance as breathing, it allows us to relate the brightness function to the time in a breathing cycle. Introducing the exponential functions for brightness variation will be the final step of code development.

Figure 3 shows the shape of the PWM output curve for linearly varying inhale and exhale phases. The equations for the inhale and exhale output are

$$v = a_{in}t + b_{in} \quad v = a_{ex}t + b_{ex} \quad (1)$$

where v is the value sent to the PWM output, a_{in} and b_{in} are the slope and intercept of the inhale function, and a_{ex} and b_{ex} are the slope and intercept of the exhale function. Remember that the values used in the `analogWrite` function must be limited to the range $0 \leq v \leq 255$.

Developing the equations for a_{in} , b_{in} , a_{ex} and b_{ex} as a function of V_{min} , V_{max} , t_1 , t_2 , t_3 and t_4 is left as an exercise.

When you implement the linear brightness variation, we recommend that you create variables like `ain`, `bin`, `aout` and `bout`, and use formulas *in your Arduino code* to assign values to those variables in terms of other variables like `Vmin`, `Vmax`, `t1`, `t2`, etc.. Figure ?? gives a hint at how to use variables instead of numerical constants.

3.2 Exponential Brightness Variation During Inhale and Exhale

To create a brightness pattern that is more natural, replace the linear ramps in intensity from Section 3.1 with other functions. For example, as shown in Figure 1, two exponential functions can be used for the inhale and exhale phases

$$v_{in} = a_{in}e^{b_{in}t} \quad v_{ex} = a_{ex}e^{b_{ex}t} \quad (2)$$

Note that the a_{in} , b_{in} , a_{ex} and b_{ex} values necessary for Equation (2) will be different from the a_{in} , b_{in} , a_{ex} and b_{ex} values for Equation (1).

3.3 Alternative Implementation with the fade Example

The standard Arduino installation includes the `Fade.ino` sketch which provides a PWM output in the form of a slow triangle wave. If the output of the `Fade` sketch is connected to an LED, the brightness of an LED will increase and decrease indefinitely. To view the code, make the following menu selections from an open Arduino window

File → Examples → Basics → Fade

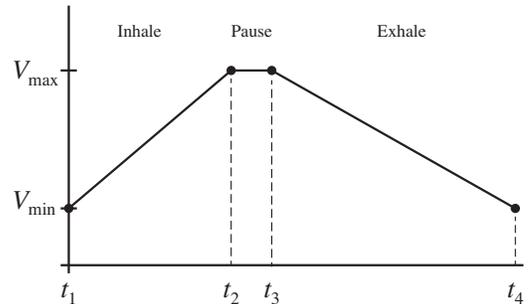


Figure 3: Simulation of breathing cycle with linear variation of brightness.

Use variables, like this:

```
int t, v;
float Vmin = 50, Vmax 255;
float Tin = ..., Tpause = ..., Tcycle = ...;

ain = ...
bin = ...
aout = ...
bout = ...

if ( t<Tin ) {
  v = ain*t + bin;
} else if ...
  ...
}

v = constrain(v, 0, 255); // to be safe
analogWrite(LED_pin, v);
```

Don't use constants like this:

```
int t, v;

if ( t<2500 ) {
  v = 1.234*t + 4.5678;
} else if ...
  ...
}

v = constrain(v, 0, 255); // to be safe
analogWrite(LED_pin, v);
```

Figure 4: Recommended coding practice for implementing the breathing LED model with variable brightness. The ... represents missing code that you will need to supply to complete the sketch.