IoT Labs

Quick Lesson

“Useful Programming Constructs for Photon”

**Table of Contents**

[Introduction](#_u3ddc2qmos3k)

[User Defined Functions - Simplify your code](#_6z6rg2c843v4)

[Timers - Code running at multiple independent time intervals](#_uzeljue997f)

[Adafruit Tutorial on Timers](#_36kv7lu2mp0o)

[Object Oriented Programming](#_2ftl0vtejiuz)

# Introduction

Contained in this Quick Lesson are a couple useful programming constructs that are worth reviewing before you start to build more complex code. These concepts include user-defined functions, non-interrupt-based timers, and a short review of object-oriented coding as applied to Photon/Arduino-like programming.

While using individual pre-made functions with delay() inside the core begin() and loop() functions will allow you to test and evaluate a basic task with a Photon/Arduino-like microcontroller, doing more than one thing at a time will require more structured coding. This Quick Lesson will help you improve your code by taking a repeated code and implementing it once in a user-defined function. It will further allow you to interleave multiple tasks over similar time frames through the use of non-interrupting timers.

# User Defined Functions - Simplify your code

It is often useful to encapsulate chunks of code into custom functions. It can make your code easier to understand and often much simpler to type. When declaring your own functions you need to tell the program what inputs (for example int or unit32) and output data types (for example int or void), as well as what actual variable to return. For quick reference, an example is included below:

|  |
| --- |
| **void loop() {**  **for(int i;i<128;i++) {**  **ShowFrame(3\*i);**  **delay(100); //wait 1sec}**  **}**  **}**  **int ThermoColorRB(int i) {**  **if (i>128) i=128;**  **if (i<0) i=0;**  **return strip.Color( i, 0, 128-i);**  **}**  **void ShowFrame(int i) {**  **strip.setPixelColor(0, ThermoColorRB(1\*i/3));**  **strip.setPixelColor(1, ThermoColorRB(2\*i/3));**  **strip.setPixelColor(2, ThermoColorRB(3\*i/3));**  **strip.show();**  **}** |

If you need a more detailed explanation there is the document [Quick Lesson - Functions, Arguments, and Return Values](https://docs.google.com/document/d/1qzRRgISDt26Y2i7srI7_TLcIXd4DDP3d13twRTbYhps/edit#) and also some decent discussions available on the Arduino website:

<https://www.arduino.cc/en/Reference/FunctionDeclaration>

# Timers - Code running at multiple independent time intervals

One of the more challenging concepts for a microcontroller is executing multiple different tasks nearly simultaneously. This is especially difficult when those tasks occur at widely different time intervals. For example, you wish to play one of two alternating notes every 2 sec, but update 3 different LEDs every 0.1 secs. One option is to use the delay() function, something like this:

|  |
| --- |
| **void loop() {**  **tone(D0,880,2000);**  **ShowFrame(3\*1);**  **delay(100);**  **ShowFrame(3\*2);**  **delay(100);**  **...**  **tone(D0,880,2000);**  **ShowFrame(3\*21);**  **delay(100);**  **ShowFrame(3\*22);**  **...**  **}** |

Obviously, this works, but it would be very difficult to change/rearrange the interleaving of these two functions. It would rapidly become totally unreasonable to operate a half-dozen functions on different schedules. The delay() function is a “blocking” function. Nothing can happen when this command is being processed. As you create more and more complicated programs you need a way to execute code at scheduled times without blocking the execution of other activities.

The function millis() returns the number of milliseconds since the Photon first connected to the cloud and started the user program. You can utilize this to control when a particular function is executed.

Here is an alternative to the code above:

|  |
| --- |
| **unsigned Timer1;**  **unsigned Timer2;**  **// \*\*\*snipped irrelevant variable setup here**  **void setup()**  **{**  **Timer1 = 0;**  **Timer2 = 0;**  **// \*\*\*snipped irrelevant procedure setup here**  **}**  **void loop() {**  **// \*\*\*snipped irrelevant loop procedure here**  **if(Timer1 < millis()) {**  **if (i>128) i=0;**  **i++;**  **ShowFrame(3\*i);**  **Timer1 = millis()+100;**  **}**  **if(Timer2 < millis()) {**  **if (TwoTone) {**  **tone(D0,440,2000);**  **} else {**  **tone(D0,880,2000);**  **}**  **TwoTone = !TwoTone;**  **Timer2 = millis()+2000;**  **}**  **}** |

There is a new feature added to the Photon after the development of above material, that implements Timers in the RTOS (Real-Time Operating System). This System Timer feature is still in development, you can see what is implemented so far in the Particle Documentation here:

<https://docs.particle.io/reference/firmware/photon/#software-timers>

## Adafruit Tutorial on Timers

Adafruit has an exceptionally good article on using object-oriented programing to implement timers for an arbitrary number of processes. Check it out here:

<https://learn.adafruit.com/multi-tasking-the-arduino-part-1/overview>

# Object Oriented Programming

Going into detail on object-oriented programming is beyond the scope of this class and documentation. It is sufficient to understand that a **class** is another data type in C programming. An **instance** of a class is a unique set of embedded data and functions. Each of the functions attached to a class is called a **method**.

This is all a fancy way of saying that object oriented programming gives us access to new functions and those functions have access to a special set of data that they need to do their job.

To use a class you must first declare an **instance** of that class. We create this instance similarly to how we declare a variable, except we use a function to set up the instance with some basic data and the methods of the class. Example of how to create and use an instance of the NeoPixel class of objects:

|  |
| --- |
| **#include "neopixel/neopixel.h"**  **Adafruit\_NeoPixel strip = Adafruit\_NeoPixel(3, D4, WS2811);**  **setup() {**  **strip.begin();**  **}**  **loop() {**  **strip.setPixelColor(0,colorBlue);**  **strip.show();**  **}** |

In the example above “strip” is an instance of the object class Adafruit\_NeoPixel. When we create this instance “strip” we have to specify the basic information needed (pin, number of pixels, and pixel type). Additionally, inside the instance “strip” there are multiple methods (functions) that allow us to do neat things. The first of these is the initialization function “strip.begin()”. Not all objects have initialization beyond the original declaration but many do.

Once the instance “strip” is fully initialized, we have access to a variety of methods (functions) that have fore-knowledge of all the initialization parameters. For example, if we call “strip.setPixelColor(0,colorBlue)”, we don’t have to specify the pin, the number of pixels, or pixel type. Also the instance “strip” stores its state (RGB value of all pixels) and any additional information it needs to do its work. This way when we declare a second instance of the NeoPixel object like so:

|  |
| --- |
| **Adafruit\_NeoPixel strip2 = Adafruit\_NeoPixel(8, D3, WS2811);** |

We can freely interchange the use of “strip.setPixelColor(0,colorBlue)” and “strip2.setPixelColor(7,colorRed)” without impacting the operation of either object.

It is this isolation of multiple similar objects that makes object-oriented programming so useful in controlling real-world devices. It also nice to use them to encapsulate code and make code easy to import. Object-oriented programing has many further advantages including inheritance, polymorphism, self-aware code, and limited recursion. See [Wikipedia](https://en.wikipedia.org/wiki/Object-oriented_programming) for more details.