# Table of Contents

## Overview
- Power and Data
- Microcontroller
- LEDs
- Cap Touch and Sensors
- Reset Button
- STEMMA QT Connector
- Lanyard Hole

## Guided Tour
- CircuitPython
  - CircuitPython Quick Start!
- Blink
  - LED Location
  - Blinking an LED
- Sound Reactive NeoPixels
  - CircuitPython Library Installation
  - Sound Reactive LED Example
- Capacitive Touch
  - CircuitPython Library Installation
  - Capacitive Touch Controlled LED Example
  - Using the BLM Capacitive Touch NeoPixel Example
- Arduino IDE Setup

## Using with Arduino IDE
- Install SAMD Support
- Install Adafruit SAMD
- Install Drivers (Windows 7 & 8 Only)
- Blink
- Successful Upload
- Compilation Issues
- Manually bootloading
- Ubuntu & Linux Issue Fix

## Adapting Sketches to M0 & M4
- Analog References
- Pin Outputs & Pullups
- Serial vs SerialUSB
- AnalogWrite / PWM on Feather/Metro M0
- analogWrite() PWM range
- analogWrite() DAC on A0
- Missing header files
- Bootloader Launching
- Aligned Memory Access
- Floating Point Conversion
- How Much RAM Available?
• Storing data in FLASH
• Pretty-Printing out registers
• M4 Performance Options
• CPU Speed (overclocking)
• Optimize
• Cache
• Max SPI and Max QSPI
• Enabling the Buck Converter on some M4 Boards

Arduino Blink
• Finding the LED

Arduino NeoPixel Rainbow
• NeoPixel
• Code

Downloads
• Files
• Schematic and Fab Print
The Black Lives Matter Education & Workshop Kit is an open-source design the Adafruit team published during the peaceful demonstrations for social justice in the summer of 2020 (https://github.com/adafruit/BLM-Badge-PCB (https://adafruit.it/RaZ)).

As a company and culture we came together to make our voices heard, share the pain we all had, the anger, and then work together for equality and justice in our communities (https://www.adafruit.com/blacklivesmatter (https://adafruit.it/N2b)). We listened to each other, we marched, we donated our time, resources, we distributed PPE at community events, we came together.
The kit is a snapshot in time, a time capsule of what we did together, and what we can build together going forward. The kit can be used for learn-to-code events remotely or in person when gatherings are safe post-COVID. We wanted to make something that would continue to emphasize the moment, is a movement.

The kits will never be for sale from Adafruit, they will be donated to learning-to-code organizations, social justice groups, and events.

This education and workshop kit features a Cortex M0+ processor that can run Arduino or CircuitPython.
• USB Type C connector - If you have only Micro B cables, [this adapter will come in handy](https://adafru.it/FQR).
• ATSAMD21E18 32-bit Cortex M0+ - 48 MHz 32 bit processor with 256KB Flash and 32 KB RAM
• Native USB supported by every OS - can be used in Arduino or CircuitPython as USB serial console, MIDI, Keyboard/Mouse HID, even a little disk drive for storing Python scripts.
• Can be used with Arduino IDE or CircuitPython
• Power with 2 x AAA batteries, rechargeable or alkaline
• Six side-lighting RGB NeoPixel LEDs
• Four Capacitive Touch pads - they can also be used as digital/analog pins
• Light Sensor
• Sound Sensor (microphone)
• Red 'pin 13' LED
• On / Off switch
• Reset switch for starting your project code over or entering bootloader mode
• Lanyard hole
Guided Tour

This board has a number of features that open up world of possibilities. This page is a tour of the board.
The USB Type C connector is used for both communicating with the microcontroller on the board, and for powering the board. Connect a USB Type C cable to install CircuitPython and edit code, or to load an Arduino sketch.

You can also power the board off of a USB battery pack. This board does not charge batteries.

The battery holder on the back takes 2 x AAA alkaline or NiMH batteries. The batteries will power the board while it's not plugged into USB to make your project portable.

The ON/OFF switch turns the board on and off. When it's to the left, the board is off, and when it's to the right, it's on. You can also use it to conserve battery power while powering the board from batteries.

If you plug your board into USB and nothing happens, make sure the switch is in the ON position!
Microcontroller

The brains of the BLM badge is the ATSAMD21E18 32-bit Cortex M0+ microcontroller. It has a 48 MHz 32 bit processor with 256KB Flash and 32 KB RAM.

LEDs

There are six side-lit addressable RGB NeoPixel LEDs around the outer edge of the board. They are accessible in CircuitPython as `board.NEOPIXEL`, and in Arduino as `PIN_NEOPIXEL`.

- There is a green LED next to the USB connector that indicates when the board is powered up.
- There is a red LED at the top-center of the board that is user controllable. It is accessible in CircuitPython as `board.D13`, and in Arduino as `13`.

©Adafruit Industries
Cap Touch and Sensors

There are four capacitive touch sensors on the board that you can use as capacitive touch inputs. These pads react to human touch! They are labeled on the back as CT1, CT2, CT3 and CT4. They are accessible in CircuitPython as `board.CAP1`, `board.CAP2`, `board.CAP3` and `board.CAP4`. They are accessible in Arduino as `1`, `2`, `3`, and `4`.

These pads are also alligator/croc-clip friendly for attaching other components or breakouts using alligator clips.

They can also be used as digital or analog pins.
There is an ambient light sensor in the middle labeled Light, which points through to the front, as seen in the second image. The light sensor is an analog input, connected to `board.LIGHT` (CircuitPython) or `A4` (Arduino) you can read it as any analog value ranging from 0 (dark) to 65535 (CircuitPython) or 1023 (in Arduino) when bright.

There is a microphone audio sensor labeled Mic. A MEMS microphone can be used to detect audio levels and even perform basic FFT functions. Instead of an analog microphone, that requires an external op-amp and level management, we've decided to go with a PDM microphone. This is a digital mic, and is a lot smaller and less expensive! You will have to use the CircuitPython/Arduino support libraries to read the audio volume, you cannot read it like an analog voltage.
**Reset Button**

There is a reset button in the middle of the board towards the top. This button lets you restart the board, or enter the bootloader (which is required to install CircuitPython).

**STEMMA QT Connector**

There is a STEMMA QT connector which allows you to connect a growing number of STEMMA QT breakouts and accessories (https://adafruit.it/Rb0) to add more capabilities to your BLM badge with no soldering needed!

**Lanyard Hole**

No badge would be complete without a lanyard hole. You can connect a lanyard or clip to this hole at the top of your board to wear it or attach it to a backpack, etc.
CircuitPython

CircuitPython (https://adafru.it/tB7) is a derivative of MicroPython (https://adafru.it/BeZ) designed to simplify experimentation and education on low-cost microcontrollers. It makes it easier than ever to get prototyping by requiring no upfront desktop software downloads. Simply copy and edit files on the CIRCUITPY drive to iterate.

The following instructions will show you how to install CircuitPython. If you've already installed CircuitPython but are looking to update it or reinstall it, the same steps work for that as well!

CircuitPython Quick Start!

Follow this quick step-by-step for super-fast Python power :) 

Download the latest version of CircuitPython for this board via circuitpython.org

https://adafru.it/QMa

Click the link above to download the latest UF2 file.

Download and save it to your desktop (or wherever is handy).
Plug your BLM kit into your computer using a known-good USB cable.

A lot of people end up using charge-only USB cables and it is very frustrating! So make sure you have a USB cable you know is good for data sync.

Double-click the Reset button on the top in the center of your board, and you will see the NeoPixel RGB LEDs around the edge of the board turn green. If they turn red, check the USB cable, try another USB port, etc. Note: The little red LED towards the top of your board will pulse red. That's ok!

If double-clicking doesn't work the first time, try again. Sometimes it can take a few tries to get the rhythm right!

You will see a new disk drive appear called BADGEBOOT.

Drag the adafruit_circuitpython_etc.uf2 file to BADGEBOOT.
The LEDs will flash. Then, the BADGEBOOT drive will disappear and a new disk drive called CIRCUITPY will appear.

That's it, you're done! :)

Blink

In learning any programming language, you often begin with some sort of Hello, World! program. In CircuitPython, Hello, World! is blinking an LED. Blink is one of the simplest programs in CircuitPython. It involves three built-in modules, two lines of set up, and a short loop. Despite its simplicity, it shows you many of the basic concepts needed for most CircuitPython programs, and provides a solid basis for more complex projects. Time to get blinky!

LED Location

The build in LED on the BLM badge is in the top-center of the back of the board, just below the first capacitive touch pad. It is labeled "L".

Blinking an LED

Save the following as code.py on your CIRCUITPY drive.

```python
# SPDX-FileCopyrightText: 2021 Kattni Rembor for Adafruit Industries
# SPDX-License-Identifier: Unlicense

import time

import CircuitPython_Blink Example - the CircuitPython 'Hello, World!'"
```

©Adafruit Industries

Page 16 of 49
import board
import digitalio

led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT

while True:
    led.value = True
    time.sleep(0.5)
    led.value = False
    time.sleep(0.5)

The built-in LED begins blinking!

Note that the code is a little less "Pythonic" than it could be. It could also be written as `led.value = not led.value` with a single `time.sleep(0.5)`. That way is more difficult to understand if you’re new to programming, so the example is a bit longer than it needed to be to make it easier to read.

It's important to understand what is going on in this program.

First you `import` three modules: `time`, `board` and `digitalio`. This makes these modules available for use in your code. All three are built-in to CircuitPython, so you don't need to download anything to get started.

Next, you set up the LED. To interact with hardware in CircuitPython, your code must let the board know where to look for the hardware and what to do with it. So, you create a `digitalio.DigitalInOut()` object, provide it the LED pin using the `board` module, and save it to the variable `led`. Then, you tell the pin to act as an `OUTPUT`.

Finally, you create a `while True:` loop. This means all the code inside the loop will repeat indefinitely. Inside the loop, you set `led.value = True` which powers on the LED. Then, you use `time.sleep(0.5)` to tell the code to wait half a second before moving on to the next line. The next line sets `led.value = False` which turns the LED off. Then you use another `time.sleep(0.5)` to wait half a second before starting the loop over again.

With only a small update, you can control the blink speed. The blink speed is controlled by the amount of time you tell the code to wait before moving on using `time.sleep()`. The example uses `0.5`, which is one half of one second. Try increasing or decreasing these values to see how the blinking changes.

That's all there is to blinking an LED using CircuitPython!
Sound Reactive NeoPixels

The BLM badge comes with six side-lit RGB NeoPixel LEDs and a sound sensor (microphone). This CircuitPython example combines those two features to have fun with sound reactive LED colors. All the hardware needed for this example is built-in to the BLM Badge, so no soldering or external components are necessary!

This section will walk you through loading the necessary CircuitPython libraries, then provide and explain a code example for sound reactive LEDs. You should have already installed CircuitPython on your board and have a CIRCUITPY drive available on your computer.

CircuitPython Library Installation

For this example, you'll need to copy two libraries to your BLM badge: NeoPixel and Adafruit Pypixelbuf.

First, download the Adafruit CircuitPython Library Bundle from circuitpython.org (https://adafruit.it/RdE). Open the resulting file, and then open the lib folder contained within.

Find the following files:

• adafruit_pypixelbuf.mpy
• neopixel.mpy

Copy the two files listed above to the lib folder on your CIRCUITPY drive.

Before you continue, make sure that the lib folder on your CIRCUITPY drive contains the adafruit_pypixelbuf.mpy and neopixel.mpy files.

Sound Reactive LED Example

Save the following as code.py on your CIRCUITPY drive.

```python
import array
import math
import board
import audiobusio
import neopixel
```
from rainbowio import colorwheel

# Increase this number to use this example in louder environments. As you increase
# the number, it increases the level of sound needed to change the color of the LEDs. 5 is good
# for quiet up to # workshop-level settings. If you plan to be in a louder setting, increase this
# number to maintain # the same behavior as in a quieter setting.
magnitude_color_modifier = 5

pixels = neopixel.NeoPixel(board.NEOPIXEL, 6, auto_write=False)
mic = audiobusio.PDMIn(
    board.MICROPHONE_CLOCK, board.MICROPHONE_DATA, sample_rate=16000, bit_depth=16
)

def normalized_rms(values):
    """Normalized Root Mean Square. Removes DC bias before computing RMS."""
    mean_values = int(sum(values) / len(values))
    return math.sqrt(sum((float(sample - mean_values) * (sample - mean_values) for sample in values)) / len(values))

audio_samples = []  # Create an empty list for sample values
while True:
    sample_array = array.array("H", [0] * 32)
mic.record(sample_array, len(sample_array))
    normalized_samples = normalized_rms(sample_array)  # Calculate normalized
    sample value
    audio_samples.append(normalized_samples)  # Add normalized values to the audio
    samples list
    audio_samples = audio_samples[-10:]  # Keep only the 10 most recent values in
    samples list
    magnitude = sum(audio_samples) / len(audio_samples)  # The average of the last
    10 audio samples
    print(magnitude)
    # Fill NeoPixels with color based on scaled magnitude
    pixels.fill(colorwheel(min(255, (magnitude / magnitude_color_modifier)))))
pixels.show()

Make a loud noise to see the LED color react!
First you import the necessary libraries and modules. Note that there are modules being imported that you did not need to copy to the lib folder in the previous step. That's because those modules are built-in to CircuitPython.

Then there is the `magnitude_color_modifier` variable. This is used to scale the sound magnitude value down to a color value. The default is 5 which is a good value for quiet settings like around the house or at your desk. If you intend to use this example in a louder setting such as a workshop or public event, you should consider increasing this number. Basically, if you see that the colors of the LEDs are not changing across the full rainbow, it may be due to an increased volume. You can increase this number to compensate.

Next is the hardware setup and the `normalized_rms` helper. This example uses the NeoPixels and the sound sensor, so you initialise these towards the beginning of your code. The helper function is used to provide a type of averaged sample sound level rather than focusing on very rapid peaks.

Before the loop, you create an empty list to hold audio sample values. This will be used in the loop.

You begin the loop by creating an array to hold 32 raw samples. Then, you use the mic to record raw signal values into the array. Next, you use the `normalized_rms` helper to normalize the values in the array.

You take the normalized sample values and add them to the `audio_samples` list. You'll use this list to create a rolling average of the normalized values. Since you are calculating a rolling average, you keep only the 10 most recent values in the `audio_samples` list at any given point in time. Then, you calculate the desired magnitude by averaging the values. This is done by adding up the 10 values (`sum(audio_samples)`) and dividing them by 10 (`len(audio_samples)`). This section of code is required because otherwise the values can be very jumpy which causes the LED color to flicker. So, we calculate a rolling average to smooth out the value and color changes.

Then, you print the `magnitude` to the serial console. Connect to the serial console to see the output if you desire. You can watch the values change as you make noise near the board.

Finally, you fill the pixels with color based on the magnitude where it ranges from 0 = red, up to green, up to blue, and back to red, with every color in between.
That's what goes into coding sound reactive NeoPixel LEDs on the BLM badge using CircuitPython!

Capacitive Touch

The BLM badge comes with six side-lit RGB NeoPixel LEDs and four capacitive touch pads. This CircuitPython example combines those two features to create touch controlled rainbow LEDs. All the hardware needed for this example is built-in to the BLM Badge, so no soldering or external components are necessary!

This section will walk you through loading the necessary CircuitPython libraries, then provide and explain a code example for touch controlled LEDs. You should have already installed CircuitPython on your board and have a CIRCUITPY drive available on your computer.

CircuitPython Library Installation

For this example, you'll need to copy two libraries to your BLM badge: NeoPixel and Adafruit Pypixelbuf.

First, download the Adafruit CircuitPython Library Bundle from circuitpython.org (http://adafruit.it/RdF). Open the resulting file, and then open the lib folder contained within.

Find the following files:

- adafruit_pypixelbuf.mpy
- neopixel.mpy

Copy the two files listed above to the lib folder on your CIRCUITPY drive.

Before you continue, make sure that the lib folder on your CIRCUITPY drive contains the adafruit_pypixelbuf.mpy and neopixel.mpy files.

Capacitive Touch Controlled LED Example

Save the following as code.py on your CIRCUITPY drive.
import time
import board
import touchio
import digitalio
import neopixel
from rainbowio import colorwheel

pixels = neopixel.NeoPixel(board.NEOPIXEL, 6, auto_write=False)
red_led = digitalio.DigitalInOut(board.D13)
red_led.direction = digitalio.Direction.OUTPUT
cap1 = touchio.TouchIn(board.CAP1)
cap2 = touchio.TouchIn(board.CAP2)
cap3 = touchio.TouchIn(board.CAP3)
cap4 = touchio.TouchIn(board.CAP4)

pixels.fill((255, 0, 0))
pixels.show()
mode = 0
touched_4 = False
color_value = 1
while True:
    if cap1.value and mode == 0:
        # Touch pad 1 to increase the brightness.
        pixels.brightness += 0.05
        pixels.show()
        time.sleep(0.15)
    if cap2.value and mode == 0:
        # Touch pad 2 to decrease the brightness.
        pixels.brightness -= 0.05
        pixels.show()
        time.sleep(0.15)
    if cap3.value and mode == 0:
        # Touch pad 3 to cycle through a rainbow of colors on the NeoPixels.
        color_value = (color_value + 1) % 255
        pixels.fill(colorwheel(color_value))
        pixels.show()
    if cap4.value and not touched_4:
        # Touch pad 4 to "disable" the other pads.
        mode += 1
        if mode > 1:
            mode = 0
        touched_4 = True
        time.sleep(0.05)
    if not cap4.value and touched_4:
        # This prevents pad 4 from spamming mode changes.
        touched_4 = False
        time.sleep(0.05)
    if mode == 0:
        # The little red LED is off when pads 1-3 are "enabled".
        red_led.value = False
    if mode == 1:
        # The little red LED is on when pads 1-3 are "disabled".
        red_led.value = True

First you import the necessary libraries and modules. Note that there are modules being imported that you did not need to copy to the lib folder in the previous step. That's because those modules are built-in to CircuitPython.

Next is the hardware set up. This example uses the NeoPixels, the little red LED and the four capacitive touch pads, so you initialise these towards the beginning of your code.
Then, you turn the pixels on red, and set up three variables needed later in the code for various reasons. The fourth pad swaps between two modes, 0 and 1, so `mode` begins at 0. Tracking whether the fourth pad has been touched begins with `touched_4 = False`. Finally, the `color_value` variable is used to cycle through the rainbow, and begins as 1.

Inside the loop, you check whether each pad was touched and what the mode is, and then do an action based on that information. There's a bit more detail to it than that, so it's best to break it down.

First, you check whether the first capacitive touch pad is touched, and if it is, AND the mode is 0, you increase the brightness of the LEDs by 0.05. Brightness is a value between 0 and 1. If brightness is already 1, nothing will happen. There is a `time.sleep(0.15)` to slow down the touch response a bit so you can more easily control the brightness.

Second, you check whether the second capacitive touch pad is touched, and if it is, AND the mode is 0, you decrease the brightness of the LEDs by 0.05. If the brightness is already 0, nothing will happen. The same `sleep()` is included here as well.

Third, you check whether the third capacitive touch pad is touched, and if it is, AND the mode is 0, you fill the NeoPixel LEDs in a rainbow cycle. As you touch the pad, the LEDs will cycle from red to green to blue and back to red.

The fourth pad toggles whether the other three are "active" or not. When you touch it the first time, the little red LED will turn on indicating that the other pads will not respond. It does this by setting `mode = 1`. Remember for the other three pads, we check whether the mode is equal to 0 before we perform the associated action. So, when the mode is set to 1, the other three pads do nothing when touched. This way you can set your LEDs to the color and brightness you want, and touch the fourth pad to "freeze" it in that state. To be able to change those settings again, touch the fourth pad again. The little red LED will turn off, indicating the other three pads are now responsive again.

When you touch a touch pad, it sends an input signal. That signal happens very quickly, and without any modification to the code, the board will receive multiple signals when you think you are only touching it once. This is desirable with regards to the rainbow cycle - you want it to continue to cycle through the colors as you touch the pad. However, if we left the fourth pad code unmodified, it would send multiple signals every time you touched it, and that means it would cycle back and forth through the two modes multiple times every time you attempted to change the mode.
You'd struggle to end up in the mode you were aiming for. Using `touched_4` to track the touch state prevents the signal spamming from happening.

The first section of code involving the fourth pad checks to see if the fourth pad has been touched AND `touched_4 = False`, and if both are valid, it adds 1 to the mode. `mode` begins at 0, so the first time you touch the pad, it changes it to 1. The second time you touch the pad, `mode` is already 1, which means `mode` is increased to 2. Since we are only dealing with two modes, 0 and 1, it checks to see if `mode` is greater than 1, and if so, sets it back to 0. This is how it toggles between two modes. Then, it sets `touched_4 = True`. This means that `if cap4.value and not touched_4` is no longer valid, so it doesn't try to execute the code beneath it. That is how it prevents the code from spamming mode changes.

The second section of code involving the fourth pad checks to see if the fourth pad is not being touched AND that `touched_4` is True. Remember, at the end of the last section of code, we set `touched_4 = True`. That means, when you remove your finger from the pad and stop touching it, this section of code will run. It sets `touched_4` back to False to reset the tracking. Now you can touch the fourth pad again to run the previous section of code and switch modes. The code ensures that you can only switch modes one at a time!

Finally, if `mode` is 0, meaning the first three touch pads are "enabled", the little red LED is turned off. If `mode` is 1, meaning the first three touch pads are "disabled", the little red LED is turned on.

That's what goes into creating a touch controlled rainbow NeoPixel LED project using CircuitPython and the BLM badge!

**Using the BLM Capacitive Touch NeoPixel Example**

To turn the brightness up, touch the first capacitive touch pad (the one closest to the lanyard clip).
To turn the brightness down, touch the second capacitive touch pad.

To cycle through the rainbow on the NeoPixel LEDs, touch the third capacitive touch pad.

To toggle reactivity on the first three touch pads, touch the fourth pad. The first time you touch it, it stops the other pads from reacting (note in the image, the rainbow doesn't cycle when the third pad is touched after the fourth pad). When you touch it again, it "re-enables" the reactivity of the other three pads. Handy for when you find a color and brightness you like, and want it to stay there!

Arduino IDE Setup

The first thing you will need to do is to download the latest release of the Arduino IDE. You will need to be using version 1.8 or higher for this guide.

Adafruit Industries
After you have downloaded and installed the latest version of Arduino IDE, you will need to start the IDE and navigate to the Preferences menu. You can access it from the File menu in Windows or Linux, or the Arduino menu on OS X.

A dialog will pop up just like the one shown below.

![Preferences dialog](Image)

We will be adding a URL to the new Additional Boards Manager URLs option. The list of URLs is comma-separated, and you will only have to add each URL once. New Adafruit boards and updates to existing boards will automatically be picked up by the Board Manager each time it is opened. The URLs point to index files that the Board Manager uses to build the list of available & installed boards.

To find the most up to date list of URLs you can add, you can visit the list of third party board URLs on the Arduino IDE wiki [here](https://adafruit.it/f7U). We will only need to add one URL to the IDE in this example, but you can add multiple URLs by separating...
them with commas. Copy and paste the link below into the Additional Boards Manager URLs option in the Arduino IDE preferences.

https://adafruit.github.io/arduino-board-index/package_adafruit_index.json

Here's a short description of each of the Adafruit supplied packages that will be available in the Board Manager when you add the URL:

- **Adafruit AVR Boards** - Includes support for Flora, Gemma, Feather 32u4, ItsyBitsy 32u4, Trinket, & Trinket Pro.
- **Adafruit SAMD Boards** - Includes support for Feather M0 and M4, Metro M0 and M4, ItsyBitsy M0 and M4, Circuit Playground Express, Gemma M0 and Trinket M0
- **Arduino Leonardo & Micro MIDI-USB** - This adds MIDI over USB support for the Flora, Feather 32u4, Micro and Leonardo using the [arcore project](https://adafruit.io/eSI).

If you have multiple boards you want to support, say ESP8266 and Adafruit, have both URLs in the text box separated by a comma (,)

Once done click OK to save the new preference settings. Next we will look at installing boards with the Board Manager.

Now continue to the next step to actually install the board support package!
Using with Arduino IDE

The Feather/Metro/Gemma/QTPy/Trinket M0 and M4 use an ATSAMD21 or ATSAMD51 chip, and you can pretty easily get it working with the Arduino IDE. Most libraries (including the popular ones like NeoPixels and display) will work with the M0 and M4, especially devices & sensors that use I2C or SPI.

Now that you have added the appropriate URLs to the Arduino IDE preferences in the previous page, you can open the Boards Manager by navigating to the Tools->Board menu.

Once the Board Manager opens, click on the category drop down menu on the top left hand side of the window and select All. You will then be able to select and install the boards supplied by the URLs added to the preferences.

Remember you need SETUP the Arduino IDE to support our board packages - see the previous page on how to add adafruit's URL to the preferences

Install SAMD Support

First up, install the latest Arduino SAMD Boards (version 1.6.11 or later)

You can type Arduino SAMD in the top search bar, then when you see the entry, click I install
Install Adafruit SAMD

Next you can install the Adafruit SAMD package to add the board file definitions.

Make sure you have Type All selected to the left of the Filter your search... box.

You can type Adafruit SAMD in the top search bar, then when you see the entry, click Install.

Even though in theory you don't need to - I recommend rebooting the IDE.

Quit and reopen the Arduino IDE to ensure that all of the boards are properly installed. You should now be able to select and upload to the new boards listed in the Tools->Board menu.
Select the matching board, the current options are:

- Feather M0 (for use with any Feather M0 other than the Express)
- Feather M0 Express
- Metro M0 Express
- Circuit Playground Express
- Gemma M0
- Trinket M0
- QT Py M0
- ItsyBitsy M0
- Hallowing M0
- Crickit M0 (this is for direct programming of the Crickit, which is probably not what you want! For advanced hacking only)
- Metro M4 Express
- Grand Central M4 Express
- ItsyBitsy M4 Express
- Feather M4 Express
- Trellis M4 Express
- PyPortal M4
- PyPortal M4 Titano
- PyBadge M4 Express
- Metro M4 Airlift Lite
- PyGamer M4 Express
- MONSTER M4SK
- Hallowing M4
- MatrixPortal M4
- BLM Badge

Install Drivers (Windows 7 & 8 Only)

When you plug in the board, you'll need to possibly install a driver
Click below to download our Driver Installer

Download Latest Adafruit Drivers package

https://adaful.it/mb8

Download and run the installer

Run the installer! Since we bundle the SiLabs and FTDI drivers as well, you'll need to click through the license

Select which drivers you want to install, the defaults will set you up with just about every Adafruit board!
Click Install to do the installin'

Blink

Now you can upload your first blink sketch!

Plug in the M0 or M4 board, and wait for it to be recognized by the OS (just takes a few seconds). It will create a serial/COM port, you can now select it from the drop-down, it'll even be 'indicated' as Trinket/Gemma/Metro/Feather/ItsyBitsy/Trellis!
Please note, the QT Py and Trellis M4 Express are two of our very few boards that does not have an onboard pin 13 LED so you can follow this section to practice uploading but you wont see an LED blink!

Now load up the Blink example

```c
// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin 13 as an output.
  pinMode(13, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
  digitalWrite(13, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);              // wait for a second
  digitalWrite(13, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);              // wait for a second
}
```

And click upload! That's it, you will be able to see the LED blink rate change as you adapt the delay() calls.

If you are having issues, make sure you selected the matching Board in the menu that matches the hardware you have in your hand.

Successful Upload

If you have a successful upload, you'll get a bunch of red text that tells you that the device was found and it was programmed, verified & reset
After uploading, you may see a message saying "Disk Not Ejected Properly" about the ...BOOT drive. You can ignore that message: it's an artifact of how the bootloader and uploading work.

**Compilation Issues**

If you get an alert that looks like

```
Cannot run program "\{runtime.tools.arm-none-eabi-gcc.path\}\bin\arm-none-eabi-g++"
```

Make sure you have installed the Arduino SAMD boards package, you need both Arduino & Adafruit SAMD board packages.


Manually bootloading

If you ever get in a 'weird' spot with the bootloader, or you have uploaded code that crashes and doesn't auto-reboot into the bootloader, click the RST button twice (like a double-click) to get back into the bootloader.

The red LED will pulse and/or RGB LED will be green, so you know that its in bootloader mode.

Once it is in bootloader mode, you can select the newly created COM/Serial port and re-try uploading.

You may need to go back and reselect the 'normal' USB serial port next time you want to use the normal upload.

Ubuntu & Linux Issue Fix

Follow the steps for installing Adafruit's udev rules on this page. (https://adafru.it/iOE)

Adapting Sketches to M0 & M4

The ATSAMD21 and 51 are very nice little chips, but fairly new as Arduino-compatible cores go. Most sketches & libraries will work but here's a collection of things we noticed.

The notes below cover a range of Adafruit M0 and M4 boards, but not every rule will apply to every board (e.g. Trinket and Gemma M0 do not have ARef, so you can skip the Analog References note!).
Analog References

If you'd like to use the ARef pin for a non-3.3V analog reference, the code to use is `analogReference(AR_EXTERNAL)` (it's AR_EXTERNAL not EXTERNAL)

Pin Outputs & Pullups

The old-style way of turning on a pin as an input with a pullup is to use

```
pinMode(pin, INPUT)
digitalWrite(pin, HIGH)
```

This is because the pullup-selection register on 8-bit AVR chips is the same as the output-selection register.

For M0 & M4 boards, you can't do this anymore! Instead, use:

```
pinMode(pin, INPUT_PULLUP)
```

Code written this way still has the benefit of being backwards compatible with AVR. You don't need separate versions for the different board types.

Serial vs SerialUSB

99.9% of your existing Arduino sketches use Serial.print to debug and give output. For the Official Arduino SAMD/M0 core, this goes to the Serial5 port, which isn't exposed on the Feather. The USB port for the Official Arduino M0 core is called SerialUSB instead.

In the Adafruit M0/M4 Core, we fixed it so that Serial goes to USB so it will automatically work just fine.

However, on the off chance you are using the official Arduino SAMD core and not the Adafruit version (which really, we recommend you use our version because it’s been tuned to our boards), and you want your Serial prints and reads to use the USB port, use SerialUSB instead of Serial in your sketch.
If you have existing sketches and code and you want them to work with the M0 without a huge find-replace, put

```c
#if defined(ARDUINO_SAMD_ZERO) && defined(SERIAL_PORT_USBVIRTUAL)
  // Required for Serial on Zero based boards
#define Serial SERIAL_PORT_USBVIRTUAL
#endif
```

right above the first function definition in your code. For example:

![Code example](image)

**AnalogWrite / PWM on Feather/Metro M0**

After looking through the SAMD21 datasheet, we’ve found that some of the options listed in the multiplexer table don’t exist on the specific chip used in the Feather M0.

For all SAMD21 chips, there are two peripherals that can generate PWM signals: The Timer/Counter (TC) and Timer/Counter for Control Applications (TCC). Each SAMD21 has multiple copies of each, called 'instances'.

Each TC instance has one count register, one control register, and two output channels. Either channel can be enabled and disabled, and either channel can be inverted. The pins connected to a TC instance can output identical versions of the same PWM waveform, or complementary waveforms.

Each TCC instance has a single count register, but multiple compare registers and output channels. There are options for different kinds of waveform, interleaved switching, programmable dead time, and so on.

The biggest members of the SAMD21 family have five TC instances with two 'waveform output' (WO) channels, and three TCC instances with eight WO channels:

- TC[0-4], WO[0-1]
And those are the ones shown in the datasheet's multiplexer tables.

The SAMD21G used in the Feather M0 only has three TC instances with two output channels, and three TCC instances with eight output channels:

- TC[3-5],WO[0-1]
- TCC[0-2],WO[0-7]

Tracing the signals to the pins broken out on the Feather M0, the following pins can't do PWM at all:

- Analog pin A5

The following pins can be configured for PWM without any signal conflicts as long as the SPI, I2C, and UART pins keep their protocol functions:

- Digital pins 5, 6, 9, 10, 11, 12, and 13
- Analog pins A3 and A4

If only the SPI pins keep their protocol functions, you can also do PWM on the following pins:

- TX and SDA (Digital pins 1 and 20)

**analogWrite() PWM range**

On AVR, if you set a pin's PWM with `analogWrite(pin, 255)` it will turn the pin fully HIGH. On the ARM cortex, it will set it to be 255/256 so there will be very slim but still-existing pulses-to-0V. If you need the pin to be fully on, add test code that checks if you are trying to `analogWrite(pin, 255)` and, instead, does a `digitalWrite(pin, HIGH)`
analogWrite() DAC on A0

If you are trying to use `analogWrite()` to control the DAC output on A0, make sure you do not have a line that sets the pin to output. Remove: `pinMode(A0, OUTPUT)`.

Missing header files

There might be code that uses libraries that are not supported by the M0 core. For example if you have a line with

```c
#include <util/delay.h>
```

you'll get an error that says

```
fatal error: util/delay.h: No such file or directory
    #include <util/delay.h>
    ^
compilation terminated.
Error compiling.
```

In which case you can simply locate where the line is (the error will give you the file name and line number) and 'wrap it' with `#ifdef`'s so it looks like:

```c
#if !defined(ARDUINO_ARCH_SAM) && !defined(ARDUINO_ARCH_SAMD) && !defined(ESP8266) && !defined(ARDUINO_ARCH_STM32F2)
#include <util/delay.h>
#endif
```

The above will also make sure that header file isn't included for other architectures

If the `#include` is in the arduino sketch itself, you can try just removing the line.

Bootloader Launching

For most other AVRs, clicking reset while plugged into USB will launch the bootloader manually, the bootloader will time out after a few seconds. For the M0/M4, you'll need to double click the button. You will see a pulsing red LED to let you know you're in bootloader mode. Once in that mode, it wont time out! Click reset again if you want to go back to launching code.
Aligned Memory Access

This is a little less likely to happen to you but it happened to me! If you’re used to 8-bit platforms, you can do this nice thing where you can typecast variables around. e.g.

```c
uint8_t mybuffer[4];
float f = (float)mybuffer;
```

You can't be guaranteed that this will work on a 32-bit platform because mybuffer might not be aligned to a 2 or 4-byte boundary. The ARM Cortex-M0 can only directly access data on 16-bit boundaries (every 2 or 4 bytes). Trying to access an odd-boundary byte (on a 1 or 3 byte location) will cause a Hard Fault and stop the MCU. Thankfully, there's an easy work around ... just use memcpy!

```c
uint8_t mybuffer[4];
float f;
memcpy(&f, mybuffer, 4)
```

Floating Point Conversion

Like the AVR Arduinos, the M0 library does not have full support for converting floating point numbers to ASCII strings. Functions like sprintf will not convert floating point. Fortunately, the standard AVR-LIBC library includes the dtostrf function which can handle the conversion for you.

Unfortunately, the M0 run-time library does not have dtostrf. You may see some references to using #include <avr/dtostrf.h> to get dtostrf in your code. And while it will compile, it does not work.

Instead, check out this thread to find a working dtostrf function you can include in your code:


How Much RAM Available?

The ATSAMD21G18 has 32K of RAM, but you still might need to track it for some reason. You can do so with this handy function:
extern "C" char *sbrk(int i);

int FreeRam () {
    char stack_dummy = 0;
    return &stack_dummy - sbrk(0);
}

Thx to [http://forum.arduino.cc/index.php?topic=365830.msg2542879#msg2542879 (https://adafru.it/m6D)] for the tip!

Storing data in FLASH

If you're used to AVR, you've probably used PROGMEM to let the compiler know you'd like to put a variable or string in flash memory to save on RAM. On the ARM, its a little easier, simply add const before the variable name:

const char str[] = "My very long string";

That string is now in FLASH. You can manipulate the string just like RAM data, the compiler will automatically read from FLASH so you dont need special progmem-knowledgeable functions.

You can verify where data is stored by printing out the address:
Serial.print("Address of str "); Serial.println((int)&str, HEX);

If the address is $2000000 or larger, its in SRAM. If the address is between $0000 and $3FFFF Then it is in FLASH

Pretty-Printing out registers

There's a lot of registers on the SAMD21, and you often are going through ASF or another framework to get to them. So having a way to see exactly what's going on is handy. This library from drewfish will help a ton!

M4 Performance Options

As of version 1.4.0 of the Adafruit SAMD Boards package in the Arduino Boards Manager, some options are available to wring extra performance out of M4-based devices. These are in the Tools menu.

All of these performance tweaks involve a degree of uncertainty. There’s no guarantee of improved performance in any given project, and some may even be detrimental, failing to work in part or in whole. If you encounter trouble, select the default performance settings and re-upload.

Here’s what you get and some issues you might encounter...

CPU Speed (overclocking)

This option lets you adjust the microcontroller core clock...the speed at which it processes instructions...beyond the official datasheet specifications.

Manufacturers often rate speeds conservatively because such devices are marketed for harsh industrial environments...if a system crashes, someone could lose a limb or worse. But most creative tasks are less critical and operate in more comfortable settings, and we can push things a bit if we want more speed.

There is a small but nonzero chance of code locking up or failing to run entirely. If this happens, try dialing back the speed by one notch and re-upload, see if it’s more stable.

Much more likely, some code or libraries may not play well with the nonstandard CPU speed. For example, currently the NeoPixel library assumes a 120 MHz CPU speed and won’t issue the correct data at other settings (this will be worked on). Other
libraries may exhibit similar problems, usually anything that strictly depends on CPU timing...you might encounter problems with audio- or servo-related code depending how it’s written. If you encounter such code or libraries, set the CPU speed to the default 120 MHz and re-upload.

Optimize

There’s usually more than one way to solve a problem, some more resource-intensive than others. Since Arduino got its start on resource-limited AVR microcontrollers, the C++ compiler has always aimed for the smallest compiled program size. The “Optimize” menu gives some choices for the compiler to take different and often faster approaches, at the expense of slightly larger program size...with the huge flash memory capacity of M4 devices, that’s rarely a problem now.

The “Small” setting will compile your code like it always has in the past, aiming for the smallest compiled program size.

The “Fast” setting invokes various speed optimizations. The resulting program should produce the same results, is slightly larger, and usually (but not always) noticeably faster. It’s worth a shot!

“Here be dragons” invokes some more intensive optimizations...code will be larger still, faster still, but there’s a possibility these optimizations could cause unexpected behaviors. Some code may not work the same as before. Hence the name. Maybe you’ll discover treasure here, or maybe you’ll sail right off the edge of the world.

Most code and libraries will continue to function regardless of the optimizer settings. If you do encounter problems, dial it back one notch and re-upload.

Cache

This option allows a small collection of instructions and data to be accessed more quickly than from flash memory, boosting performance. It’s enabled by default and should work fine with all code and libraries. But if you encounter some esoteric situation, the cache can be disabled, then recompile and upload.

Max SPI and Max QSPI

These should probably be left at their defaults. They’re present mostly for our own experiments and can cause serious headaches.
Max SPI determines the clock source for the M4’s SPI peripherals. Under normal circumstances this allows transfers up to 24 MHz, and should usually be left at that setting. But...if you’re using write-only SPI devices (such as TFT or OLED displays), this option lets you drive them faster (we’ve successfully used 60 MHz with some TFT screens). The caveat is, if using any read/write devices (such as an SD card), this will not work at all...SPI reads absolutely max out at the default 24 MHz setting, and anything else will fail. Write = OK. Read = FAIL. This is true even if your code is using a lower bitrate setting...just having the different clock source prevents SPI reads.

Max QSPI does similarly for the extra flash storage on M4 “Express” boards. Very few Arduino sketches access this storage at all, let alone in a bandwidth-constrained context, so this will benefit next to nobody. Additionally, due to the way clock dividers are selected, this will only provide some benefit when certain “CPU Speed” settings are active. Our PyPortal Animated GIF Display (https://adafru.it/EkO) runs marginally better with it, if using the QSPI flash.

Enabling the Buck Converter on some M4 Boards

If you want to reduce power draw, some of our boards have an inductor so you can use the 1.8V buck converter instead of the built in linear regulator. If the board does have an inductor (see the schematic) you can add the line `SUPC->VREG.bit.SEL = 1;` to your code to switch to it. Note it will make ADC/DAC reads a bit noisier so we don’t use it by default. You’ll save ~4mA (https://adafru.it/F0H).

Arduino Blink

In learning any programming language, you often begin with some sort of Hello, World! program. In electronics, Hello, World! is blinking an LED. Blink is one of the simplest programs in Arduino. Despite its simplicity, it shows you many of the basic concepts needed for most Arduino programs, and provides a solid basis for more complex projects. Time to get blinky!
Finding the LED

The little built-in red LED (indicated by the red box in the image) is located towards the top of the board in the center. It is labeled "L" on the board.

First open the Arduino IDE and create a new sketch by clicking File > New

Then, copy the code shown below and paste it into the new sketch.

```cpp
void setup() {
  pinMode(LED_BUILTIN, OUTPUT);
}
void loop() {
  digitalWrite(LED_BUILTIN, HIGH);
  delay(1000);
  digitalWrite(LED_BUILTIN, LOW);
  delay(1000);
}
```

The little red LED begins blinking!

First, you set up the LED on pin LED_BUILTIN and set the pin to be an output.

Then, inside the loop, you first turn the LED on (HIGH is the voltage level), and wait for 1 second (1000ms). Then you turn off the LED by setting the voltage low, and wait for another second.

Try changing the delay to see how it affects the blink speed.

That's all there is to blinking an LED with Arduino!
Arduino NeoPixel Rainbow

The BLM Badge comes with six side-lit NeoPixel LEDs around the edge of the board. These RGB NeoPixel LEDs can be used to create all the colors of the rainbow and everything in between. The following example displays a flowing rainbow across all the LEDs.

**NeoPixel**

You'll need to install one library in Arduino for this example to work. Open the Arduino IDE. Click Arduino > Sketch > Include library... > Manage Libraries, then type 'neopixel' in the search box. Install the library shown here.

![Library Manager Image]

**Code**

In the Arduino IDE, create a new sketch by clicking File > New

Then, copy the code shown below and paste it into the new sketch.

```c
#include <Adafruit_NeoPixel.h>

#define LED_COUNT 6
#define LED_PIN PIN_NEOPIXEL

Adafruit_NeoPixel strip(LED_COUNT, LED_PIN, NEO_GRB + NEO_KHZ800);

void setup() {
    strip.begin();
    strip.show();
    strip.setBrightness(50);
}

void loop() {
    rainbow(10);
}

void rainbow(int wait) {
    for(long firstPixelHue = 0; firstPixelHue < 5*65536; firstPixelHue += 256) {
        for(int i=0; i&lt;strip.numPixels(); i++) {
            int pixelHue = firstPixelHue + (i * 65536L / strip.numPixels());
            strip.setPixelColor(i, strip.gamma32(strip.ColorHSV(pixelHue)));
        }
        strip.show();
    }
}
```

©Adafruit Industries
Rainbows!

First you include the NeoPixel library you installed.

Next, you tell the code you have 6 NeoPixels, and the pin they are on. Then, you declare the `NeoPixel` object using the info you provided.

In `setup`, you initialise the `NeoPixel` strip object, make sure the LEDs begin turned off, and set the brightness to approximately 1/5 maximum (max is 255). These LEDs are really bright at max!

In the `loop`, we run the rainbow code with a 10ms delay. Increase this number to slow down the rainbow.

Finally, we have the `rainbow` code, which requires you to provide a wait time in milliseconds.

That's all there is to making NeoPixel rainbows with Arduino and the BLM badge!

## Downloads

### Files

- [ATSAMD21 Datasheet](https://adafruit.it/xZe)
- [Webpage for the ATSAMD21E18 (main chip used)](https://adafruit.it/xZf)
- [EagleCAD PCB files on GitHub](https://adafruit.it/RaZ)
- [Fritzing object in the Adafruit Fritzing Library](https://adafruit.it/Ra-)