State And Events In CircuitPython

Josh Johnson @ TriPython

11/29/2018

Follow Along With Code Samples!

https://jjmojojjmojo.github.io/circuitpython_code_samples/

https://github.com/jjmojojjmojo/circuitpython_code_samples

More Background, Ongoing Blog Post

https://jjmojojjmojo.github.io/circuitpython-state-part-1.html

Who Am I?

- Josh Johnson aka jjmojojjmojo
- Programmer since 1999
- Pythonista since… forever.
- http://jjmojojjmojo.github.io
- @jjmojojjmojo
- jjmojojjmojo@gmail.com
- ***Not an electrical engineer!***

What Is This Talk?

- This talk is based on an (ongoing) blog series.
- We'll be digging into some electronics principals as they relate to microcontrollers, and we'll be using Python.
- It will summarize and skim, but cover the major topics.

Who Is This Talk For?

This talk is designed for people who:

- Understand *basic* Python syntax.
- Have used microcontroller development boards, in a casual way

-- OR –

- Pick up syntax contextually.
- Know nothing about microcontrollers and want an easy way to get into the scene.

Topics Covered

In this talk we'll dive into:

- State – tracking it and ways to reason about it.
- Events – how to detect them and act.
- Applying state and events to projects.
- Basic Object-Oriented Programming (OOP) concepts:
  - Classes
  - Polymorphism
  - Abstraction
- Advanced OOP:
  - Dependency Injection
  - Proxy Pattern

Rough Topic Overview

1. The Platform. **x**
2. Setup/Construction **x**
3. Programming Philosophies:
    a) State
    b) Event Detection
    c) Polymorphism
    d) Dependency Injection
4. Example Applications – UNFINISHED! 😳

The Platform

What Is CircuitPython?

- A fork of MicroPython, for the **ATSAMD21** and **ATSAMD51** ARM-based processors.
- Created and driven by Adafruit.
- It's a full-featured, but stripped down Python implementation
- It's aimed at beginners.

What Is The M0 And M4 Platform?

- Adafruit has created a suite of development boards based on the **ATSAMD21** and **ATSAMD51** processors (respectively).
- The boards come in many shapes and sizes.
- Common features:
  - Arduino IDE compatible.
  - Run at 3.3volts.
  - Onboard red LED on pin 13
  - Onboard RGB LED
  - Green power LED
  - Reset switch
  - USB – used for console access, power, and programming.
  - Any pin can be analog or digital
  - Most boards support capacitive touch (up to 7 pins depending on the board)

M0 vs M4

- M0 = ATSAMD21
  - **48 MHz 32-bit ARM Cortex-M0+**
  - **256KB Flash** (primary storage for python scripts)
  - **32 KB RAM**
- M4 = ATSAMD51
  - **120 MHz 32-bit ARM Cortex M4**
  - **512 KB Flash**
  - **192 KB RAM**
  - No capacitive touch.

Express Vs ”Regular”

- Express models include 2MB of extra SPI-based flash memory.
- The memory “just works”.
- Recommendation: start with an express board!

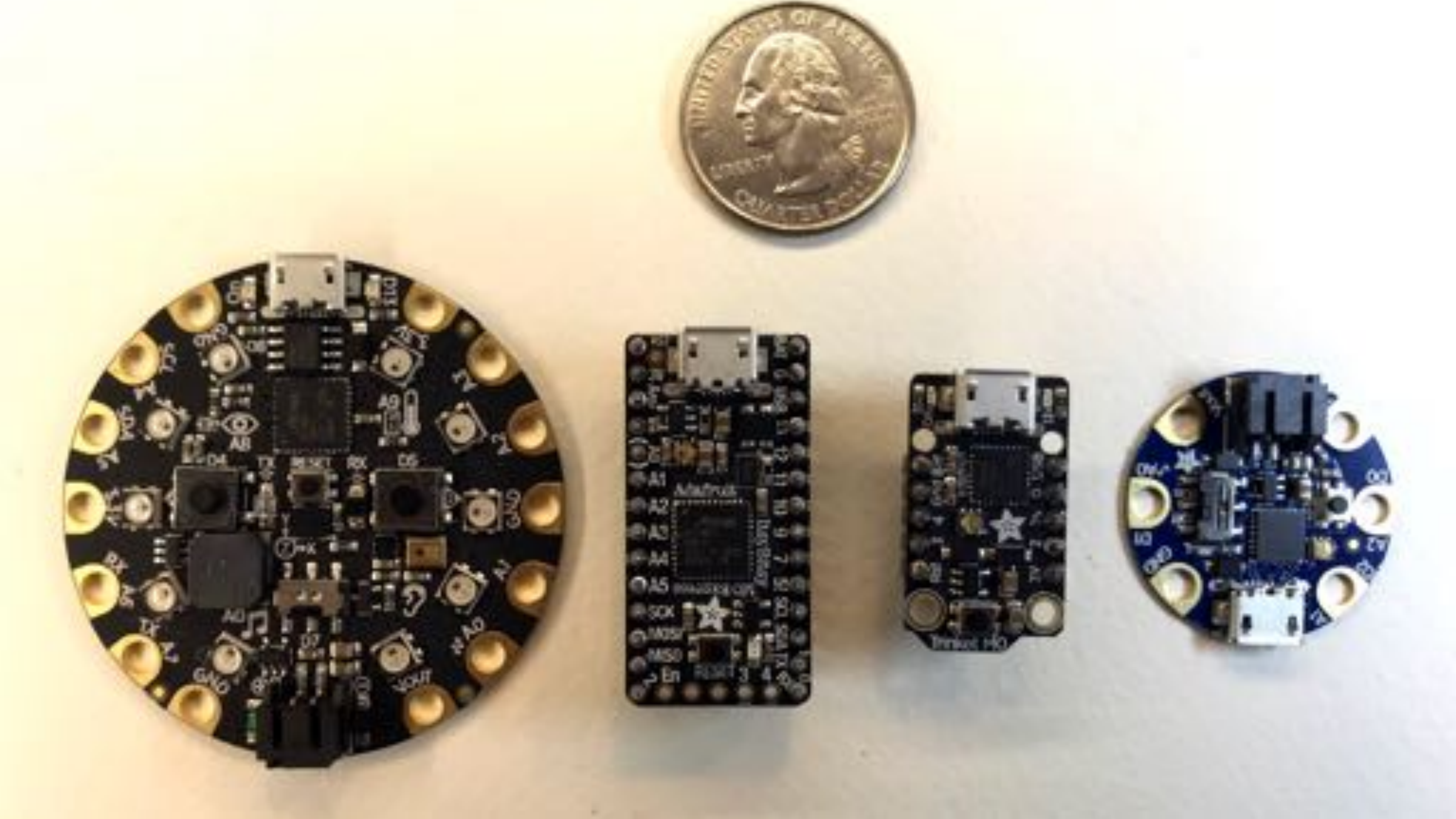
The Platform: What's Great

- Affordable, well-supported development boards.
- Lots of great options.
- Tons of built-in peripherals, powerful microcontrollers.
- Lots of great documentation.
- No dedicated computer or installed software needed.
- Mu is there to help make the process even easier.
- You can progress from CircuitPython to the Arduino IDE as you get more sophisticated.
- Under constant development.

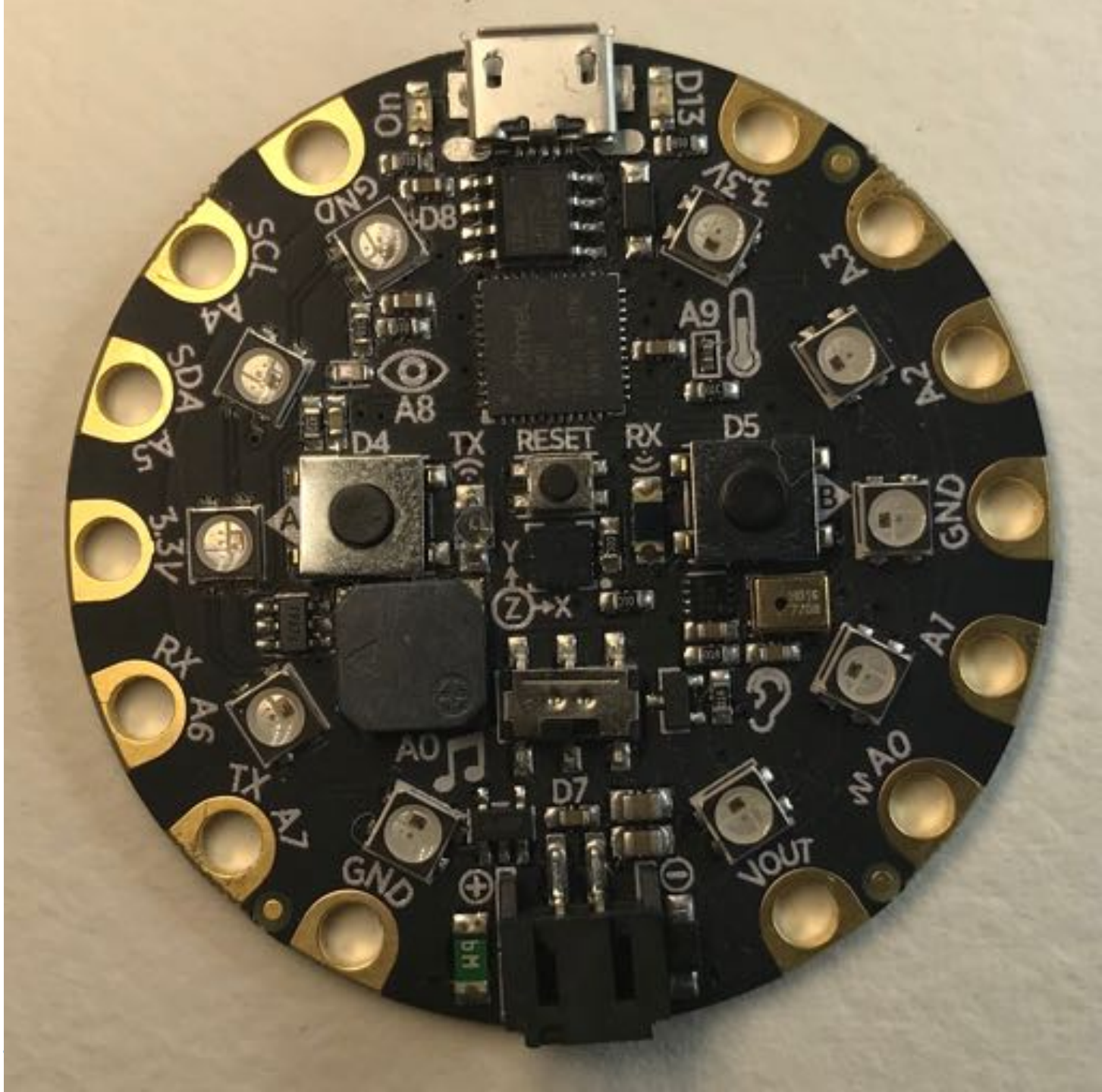
The Platform: What's Not So Great

- There are some minor differences between CircuitPython and MicroPython.
- Missing some microcontroller features.
- Documentation beyond the basics is fragmented or non-existent.
- Support for things Adafruit doesn't sell is spotty.
- It's hard to see how things work behind the libraries.
- Python is slow.

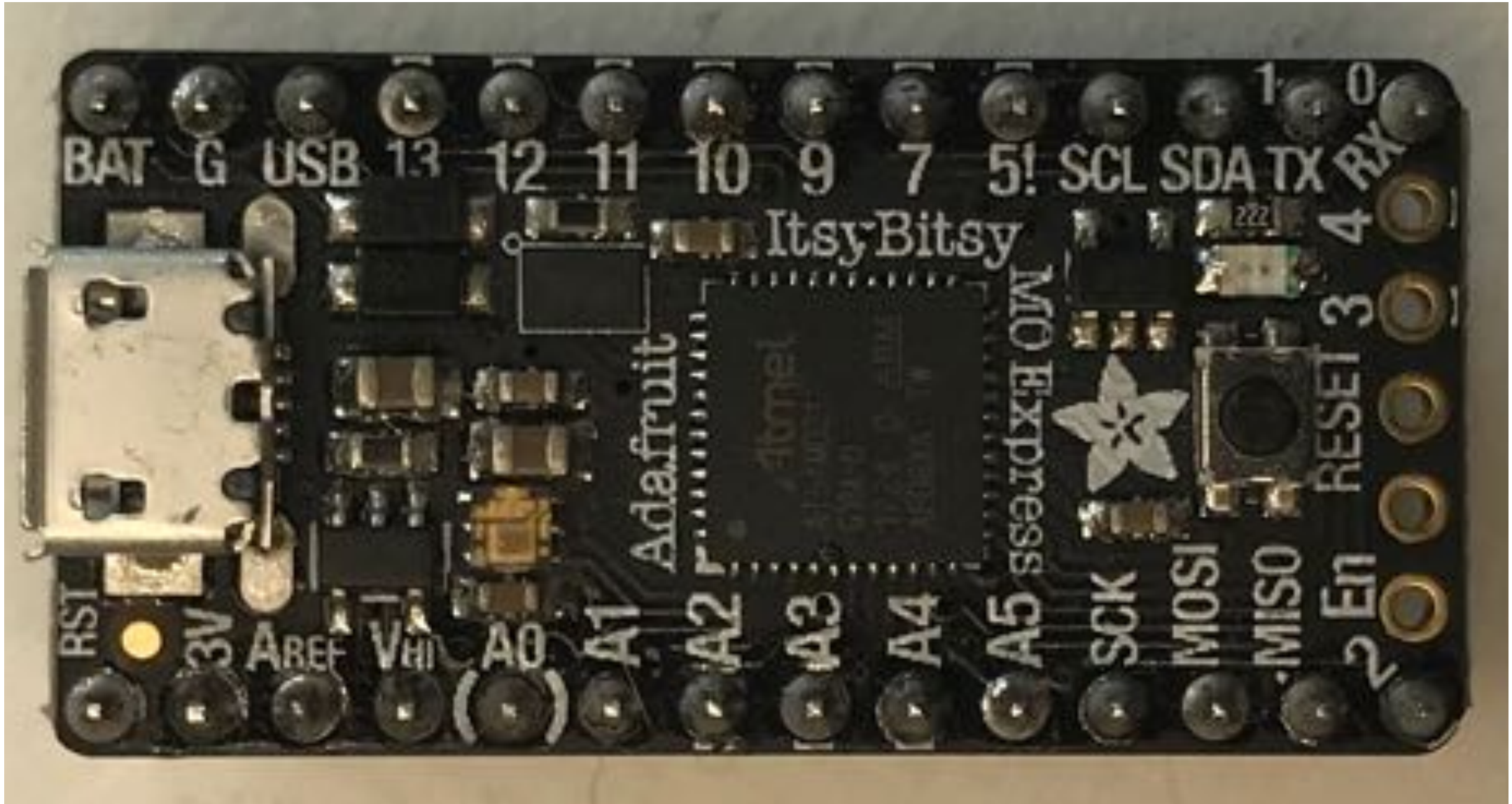
My Boards


RESET
D4
D5
TX
RX
A8
A9
GND
VOUT
A1
A2
A3
A4
A5
SCK
MOSI
MISO
En

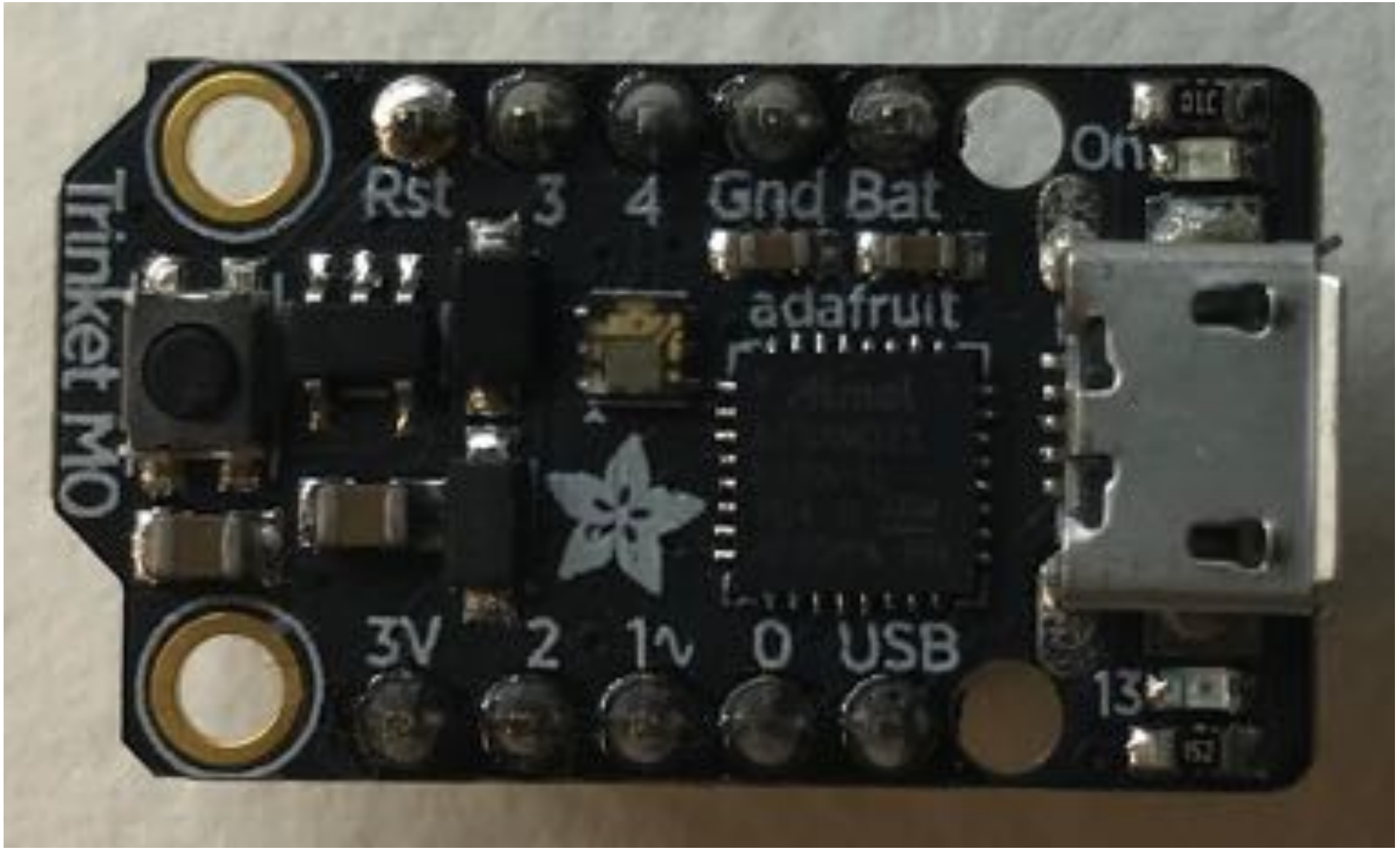
The CircuitPlayground M0 Express



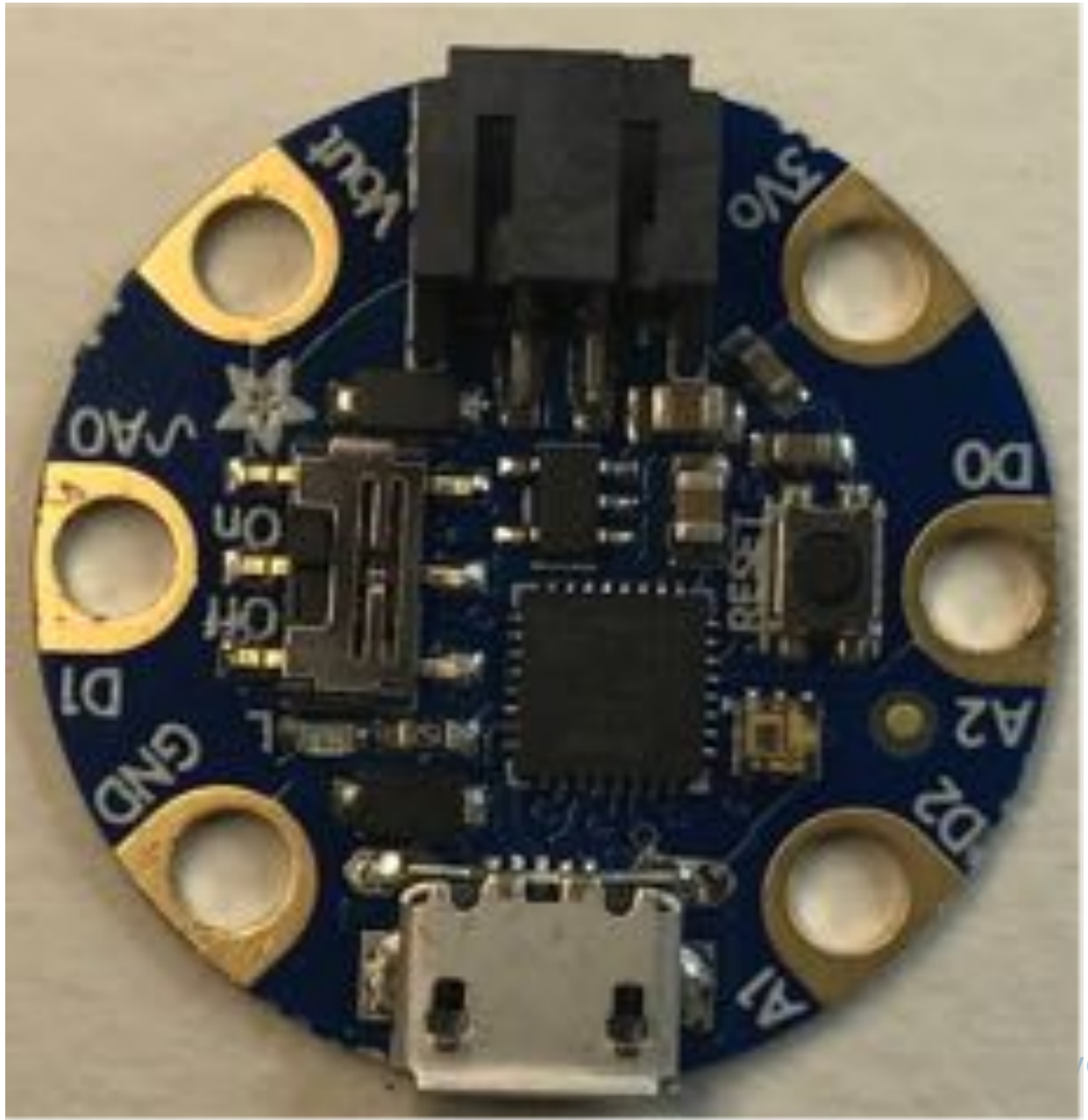
ItsyBitsy M0 Express



Trinket M0



GEMMA M0


VBAT
3V0
A0
D0
On
Off
RESET
D1
A2
GND
D2
A1


ADAFRUIT
GEMMA M0
PyCon 2018

Getting Set Up

Materials List

Basics

- A M0 or M4 board. I Recommend starting with the CircuitPlayground Express or the Metro M0 Express.
- A micro-usb cable for charging, power, and programming.
- Access to a computer with a free USB port.
- Mu

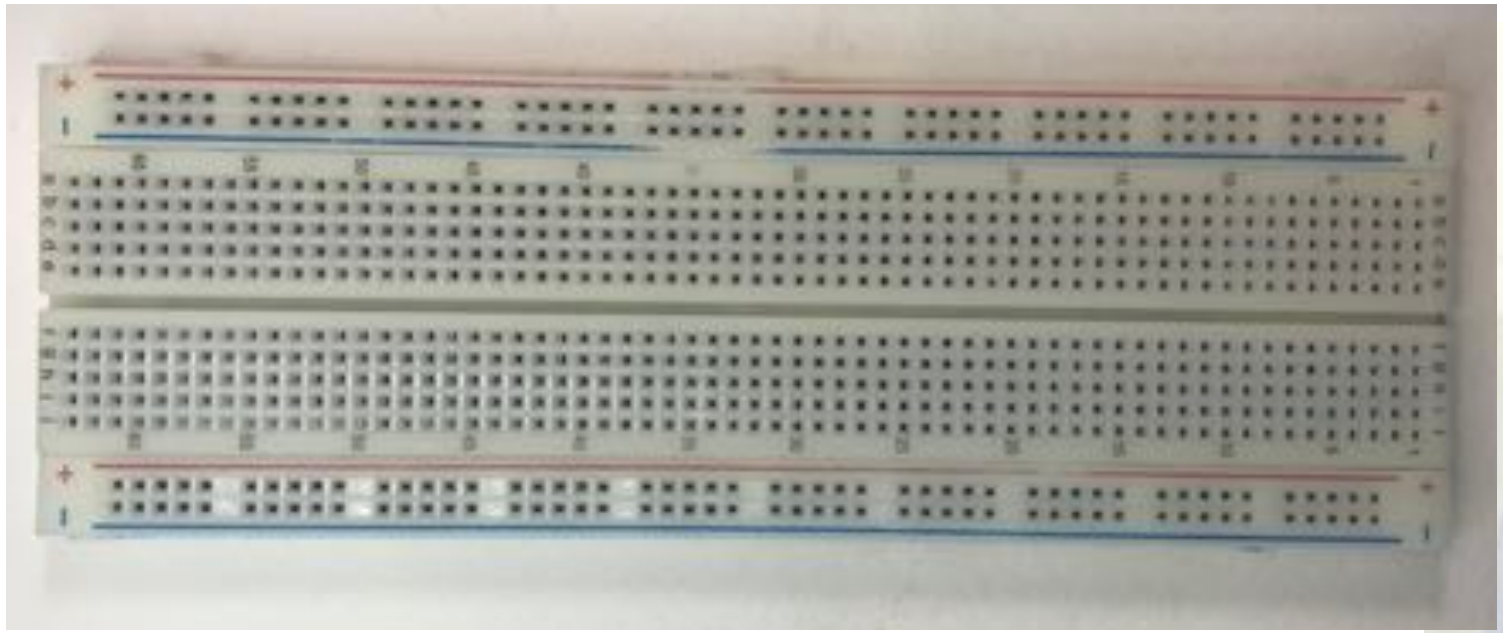
Optional, but a good idea

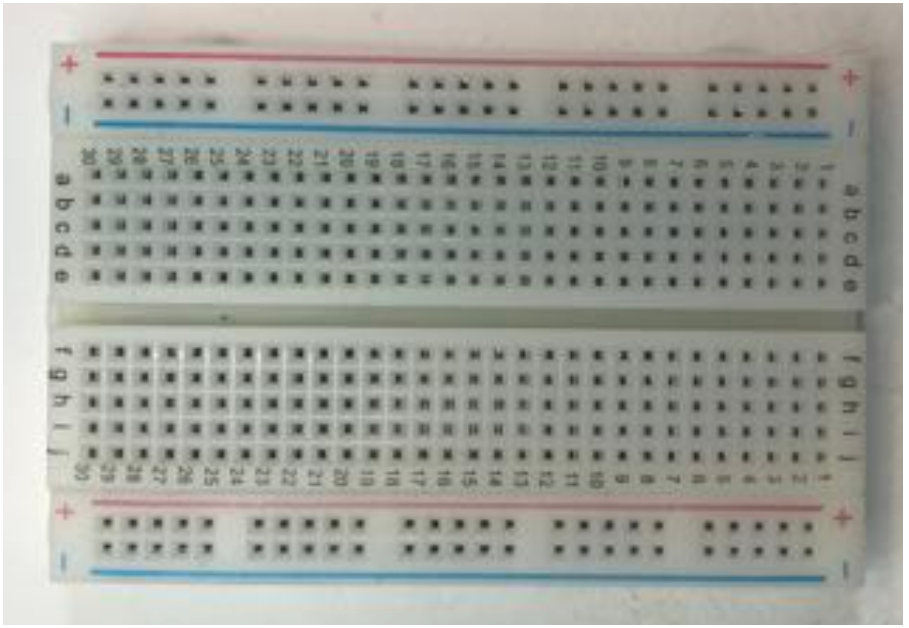
- If the headers are not attached, you will need a soldering iron, solder, and tip tinner. A solder wick is not a bad idea as well.
- Wire cutters for trimming component leads and cutting your own jumper cables.
- A wire stripper, if you are using spools of hookup wire.
- Tweezers for pulling things out of breadboards.
- Small plyers for bending wire and component leads.
- A digital multimeter.

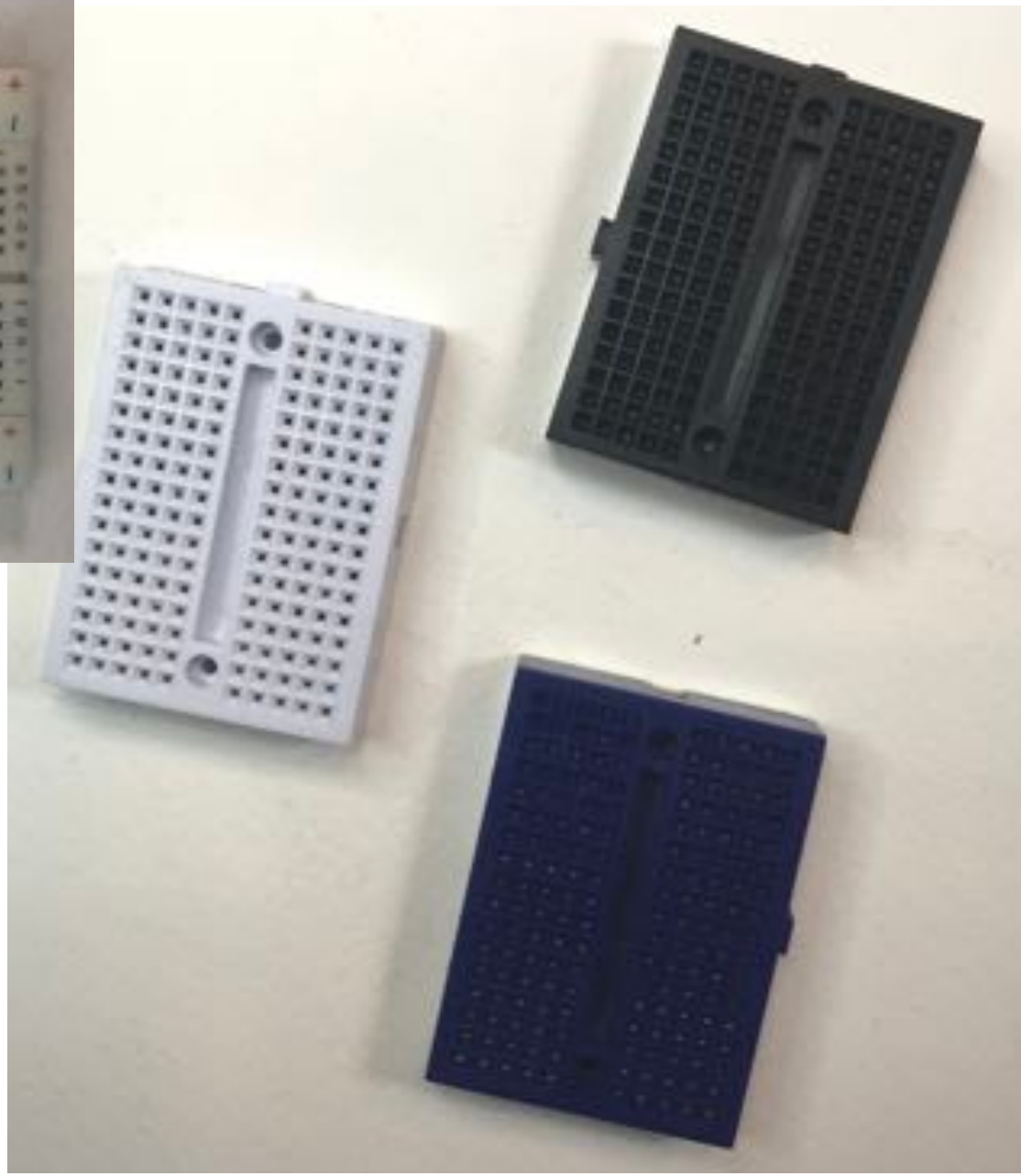

GDT-11

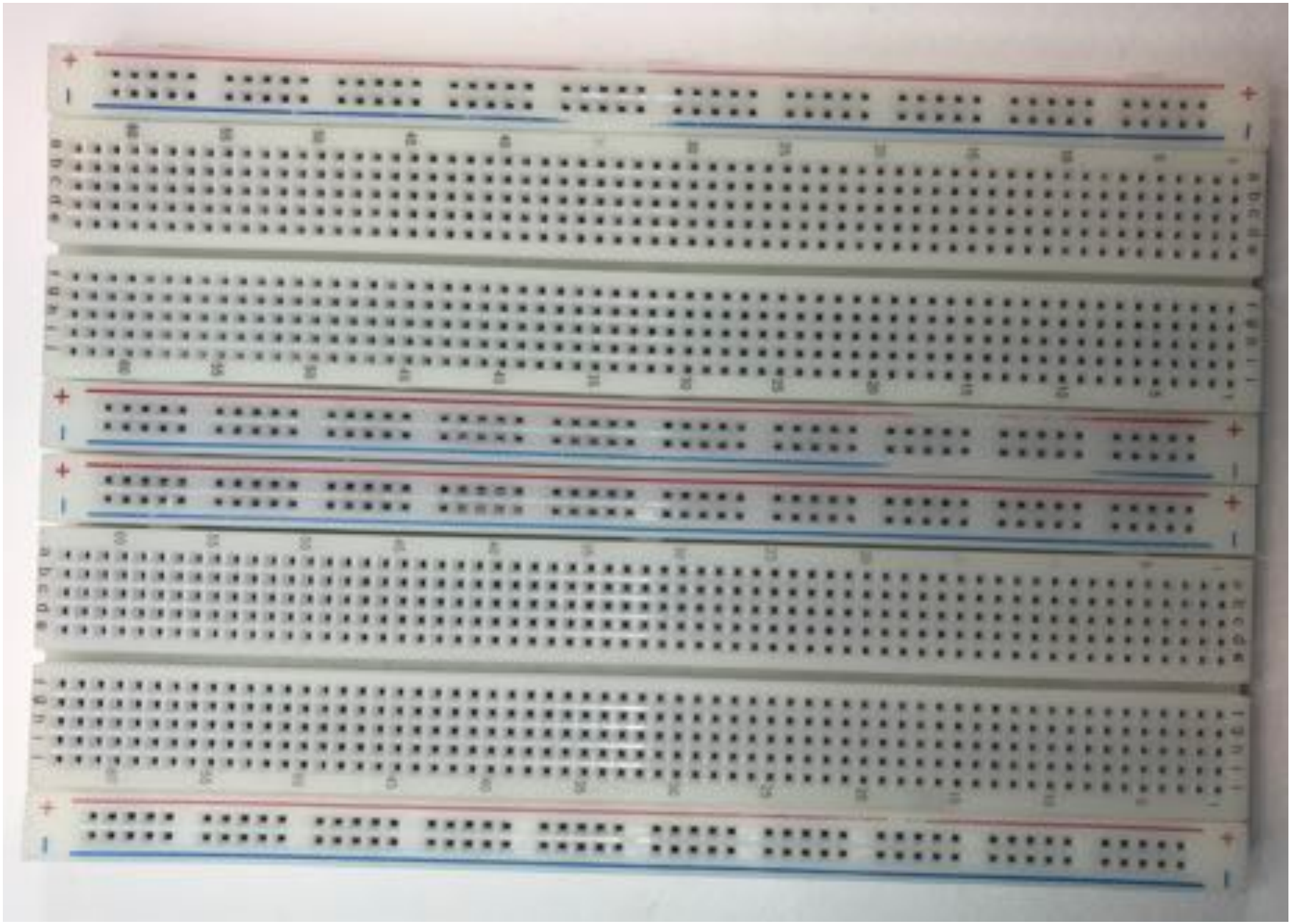
Breadboards

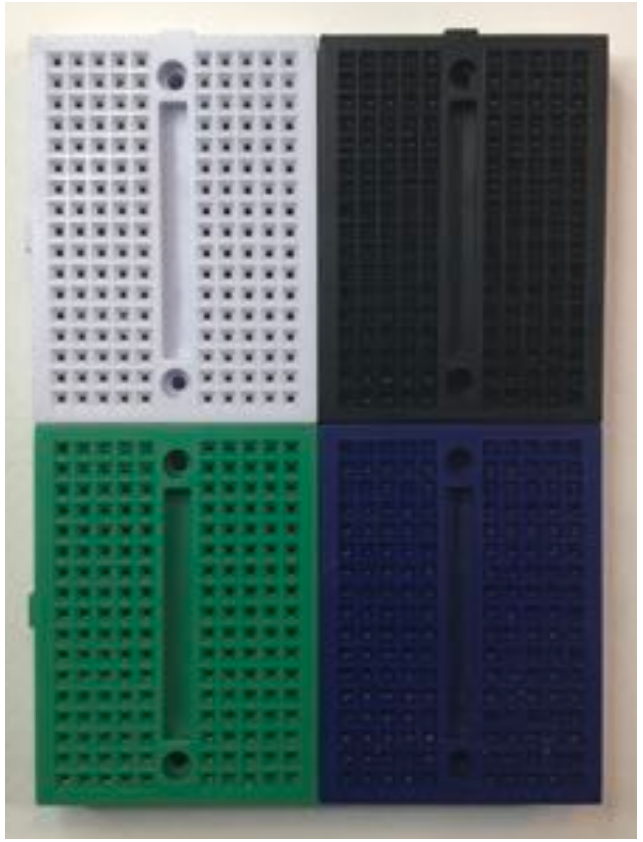
- Any kind will do.
- Larger ones are better for more complex projects.
- Most breadboards are modular!





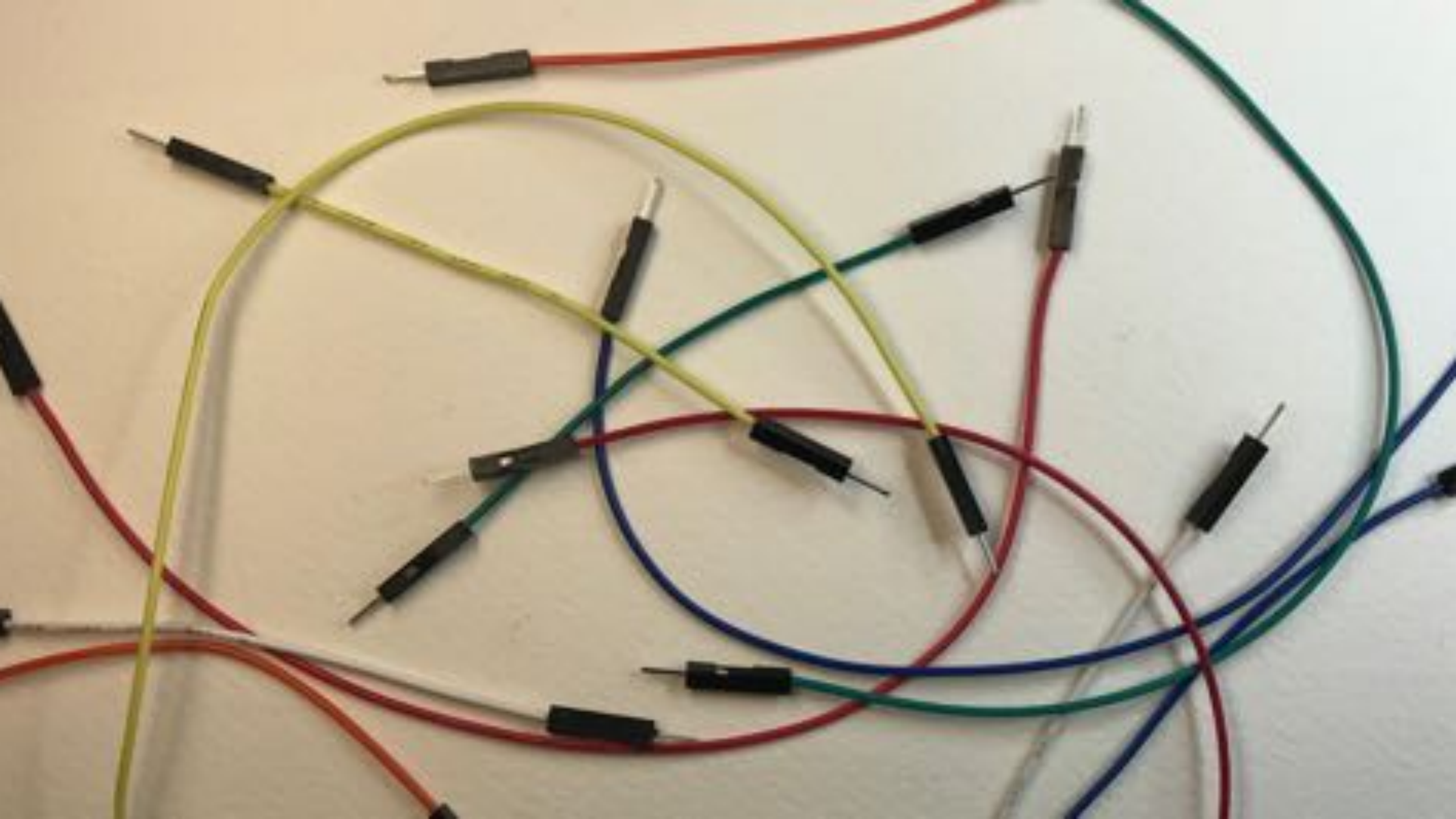


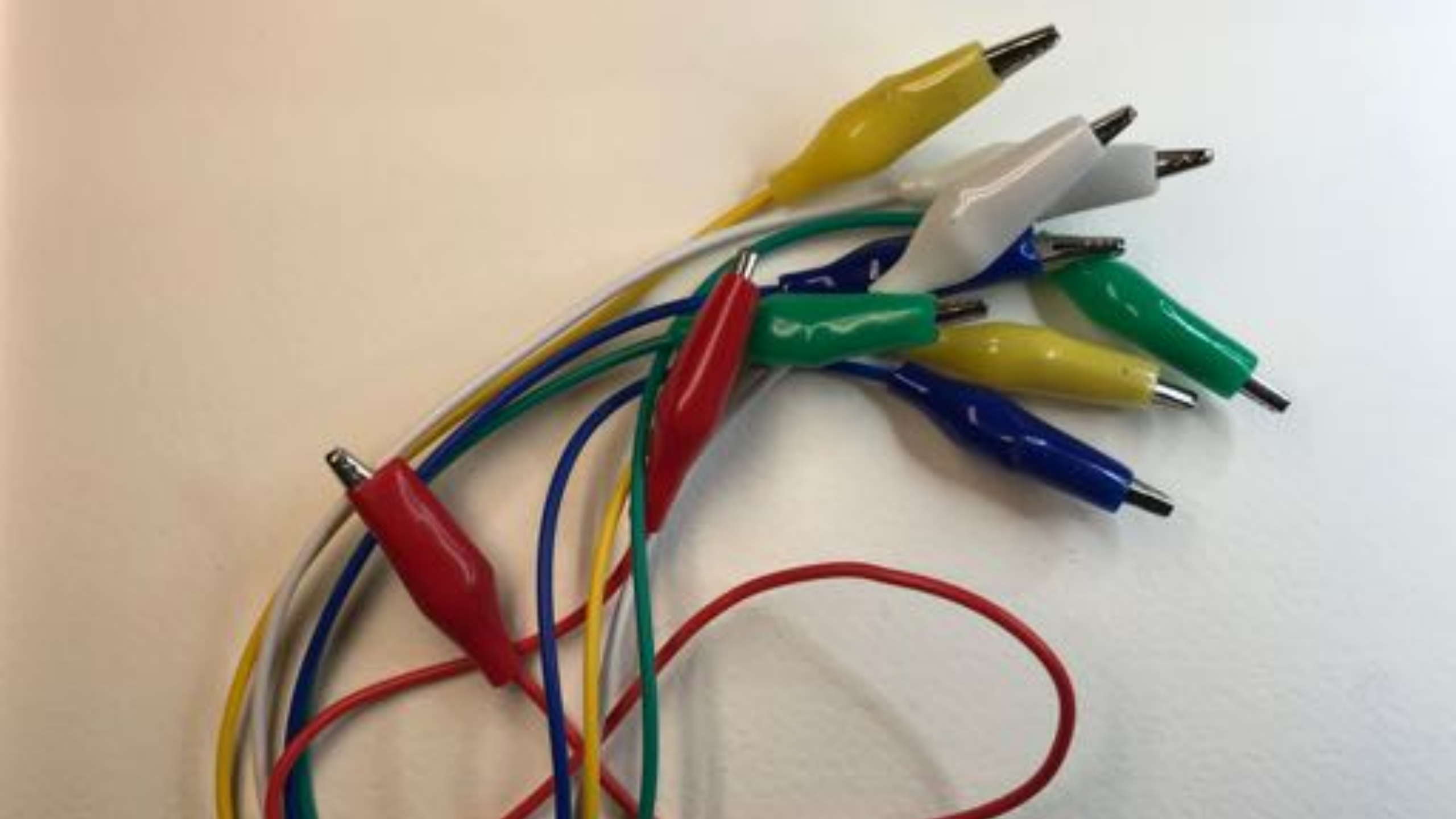




Connectors

- 22 Gauge solid core hook-up wire
- Alligator clips
- Jumper wires (male to male)
- Alligator-to-jumper wires





Buttons

- Any kind of momentary switch will work.
- Try to find breadboard-friendly.

Thermistor

- 10kΩ Precision Epoxy Thermistor - 3950 NTC
- (1) 10kΩ resistor

Photocell

- CdS photoresistor
- (1) 10kΩ resistor

Dev Board Preparation

Update To The Latest CircuitPython

Definitive Guide

Basic Overview:

1. Download the UF2 file for your board.
2. Double-click the reset button – the red LED will pulse.
3. Drag the UF2 onto the *BOOT drive that appears.
4. Wait a few seconds.

Download/Update Libraries

Docs/Offical Overview

Basic Overview

- Remove any old `*.mpy` files in the `lib` directory on your `CIRCUITPY` drive
- Download the bundle for your version from github.
- Unzip - this will create a local `lib` directory.
- Find the libraries you need and copy them over to your `CIRCUITPY/lib` folder.

Libraries Used In This Talk

- adafruit_dotstar
- neopixel
- adafruit_thermistor

Electronics

MCU IO Explained!

MCU I/O Briefly Explained

- Microcontroller Units (MCUs), in broad terms, are collections of discreet, microscopic electronic components that run at a specific voltage.
- MCUs have a core that loads and runs instructions into/from RAM.
- MCUs have components used as inputs and outputs, to interact with the outside world (I/O, GPIO)
- I/O can be *digital* or *analog.*
- The components that make up I/O include transistors, resistors, analog-to-digital converters (DACs), and digital-to-analog converters (ADCs).

Inputs: Digital Vs Analog

- Analog inputs read a variable voltage from [some minimum] to [some maximum] and use an ADC to convert it into a number within a given range.
- Digital inputs read a fixed voltage. If it's [below some minimum], the input reads as "LOW", and if it's [above some minimum] the input reads "HIGH"

Outputs: Digital vs Analog

- Digital outputs are like buttons in a way – you turn them ”on” to provide power, and turn them “off” to disconnect.
- Analog outputs create a variable voltage through a digital-to-analog converter (DAC).

Digital Inputs: Buttons

Button Basics

- A button (switch) is a component that something (usually a person) interacts with to complete a circuit.
- *Momentary* switches complete the circuit while the button is pressed, and break it when it is not.
- Other switches (slide, toggle, etc). Complete the circuit when put into the “on” position, and it stays complete until the switch is put into the “off” position.

Button Wiring

- Buttons can be wired to provide power (button is connected to a positive voltage), or to drain power (button is connected to ground).
- The easiest (and arguably safest) way to wire buttons to a MCU is in the "draining" style.
- The difference is that when wired this way, the button will read "LOW" when its *actuated*, which can be counter-intuitive, since this maps to False in Python.

Pull-up/Pull-down

- Inputs, by default ”float” – their voltage is looking for somewhere to go, so the readings will be all over the place.
- We encourage the voltage to go where we want (ground, main power) using a *resistor.*
- The resistors are referred to as *pull-down* and *pull-up* depending on how the input is wired.
- You can add your own resistors, however most MCUs provide built-in resistors for each input you can actuate in software.

In Our Demo Circuits: Buttons

- The buttons are all wired to ground, and use the built-in pull-up resistors.
- The built-in buttons on the CircuitPlayground Express are wired differently to make the inputs more intuitive for new programmers, and are configured to use the pull-down resistors.

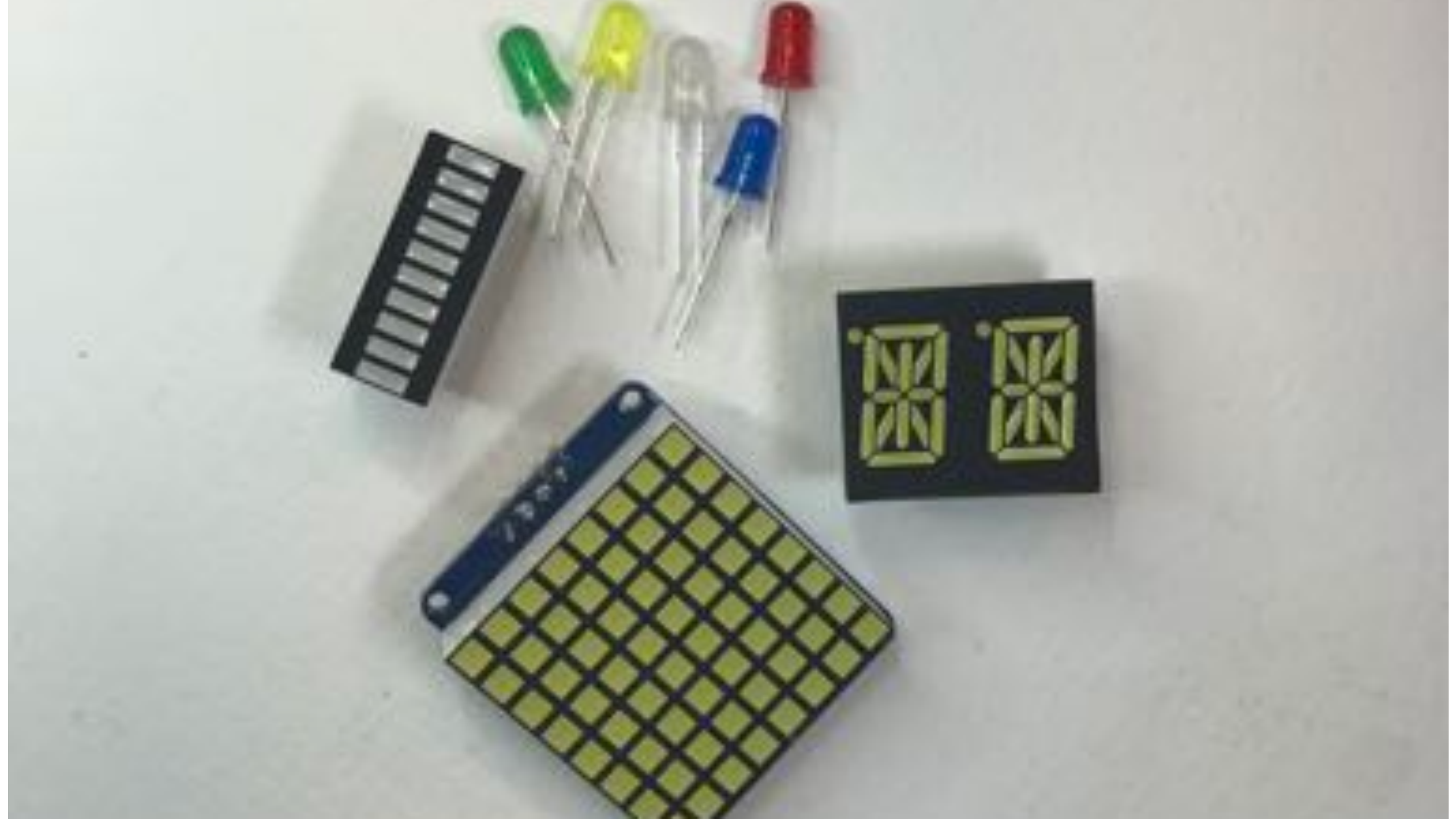
Digital Outputs: LEDs

LED Basics

- LED = Light Emitting Diode.
- Diodes only let electricity move in one direction.
- LEDs emit photons as electricity moves through them.
- Different wavelengths of light are produced by different chemicals and elements in the diode (it tends to be a fairly narrow band).
- Pure white light is produced in two ways:
  - Combining Red, Green and Blue LED light.
  - Coating ultra-violet LEDs in phosphorescent chemicals.

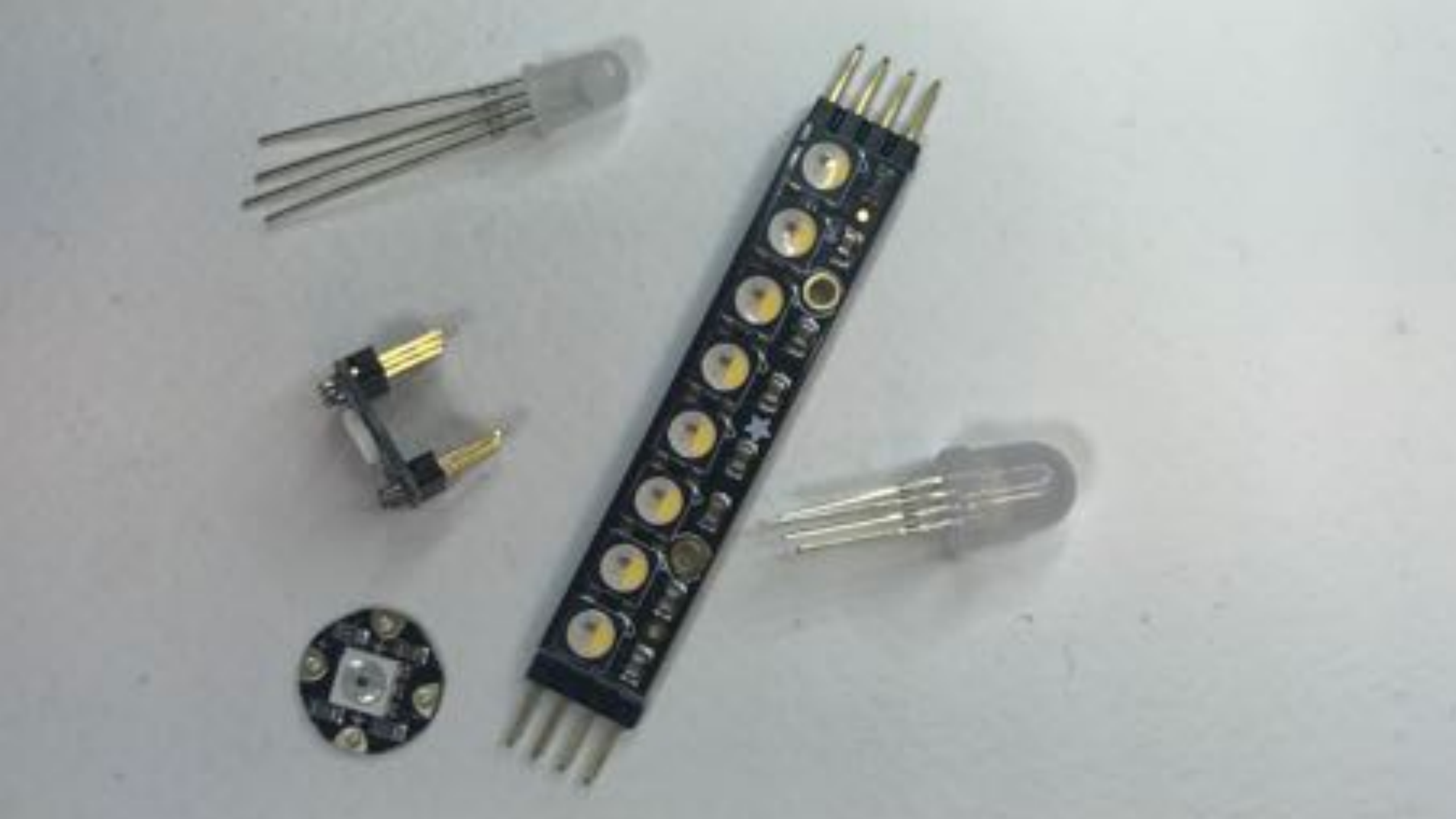
‘Simple’ LEDs

- To prevent the LED from taking all the power it can and burning out, all simple LEDs have resistors wired to them.
- Simple LEDs are just on or off, so they are connected to digital outputs.
- To dim a simple LED, a technique known as pulse-width-modulation (PWM) is used.



RGB LEDs

- RGB stands for Red, Green, and Blue – there are actually three LEDs in one package.
- You can create nearly any color of light in the visible spectrum by changing the brightness of each LED component.
- All on = White light
- The RGB LEDs on our boards are special – they contain a controller chip so they can be chained and individually controlled through one or two IO pins. They also handle PWM internally so we don't have to.

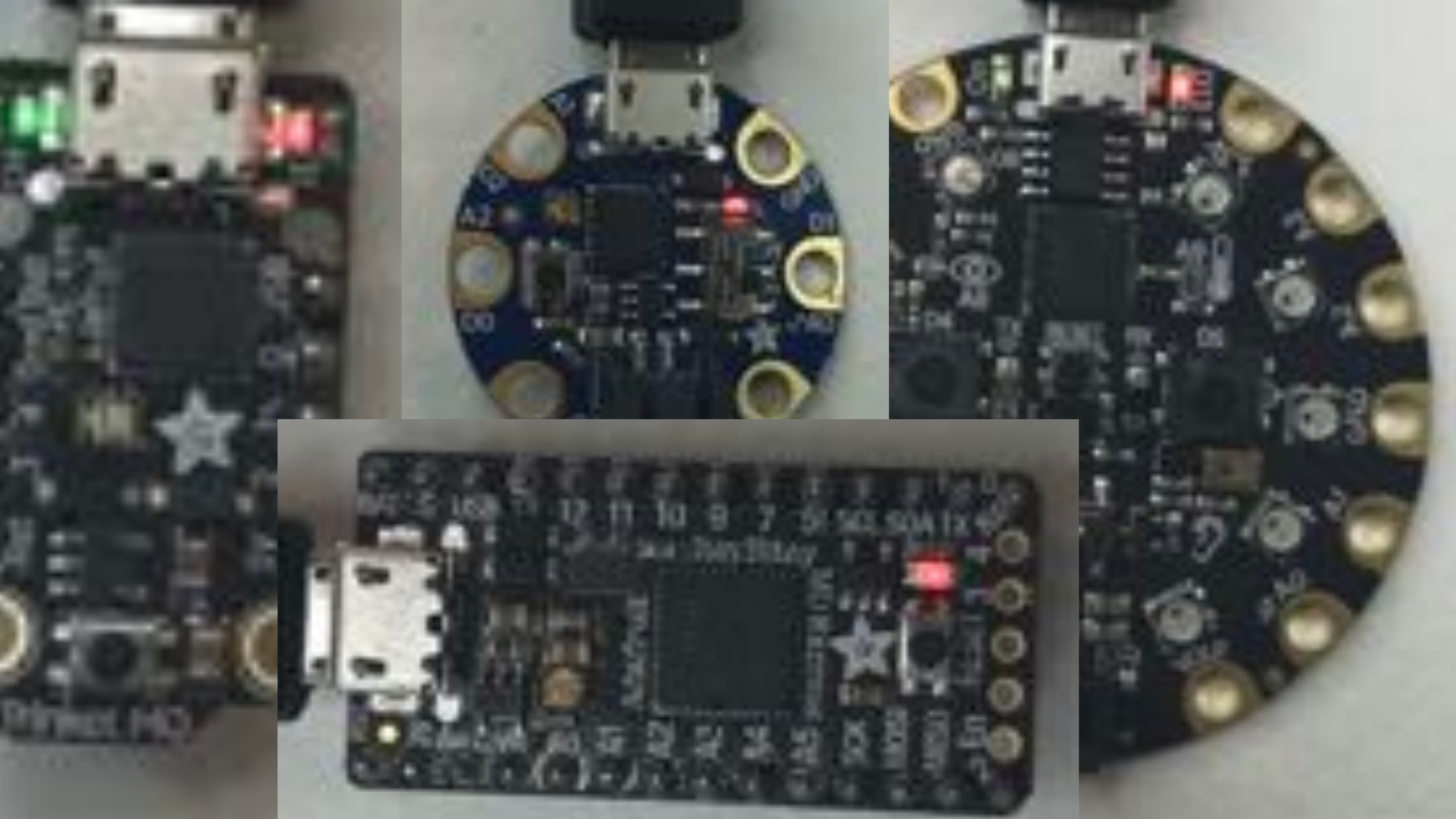


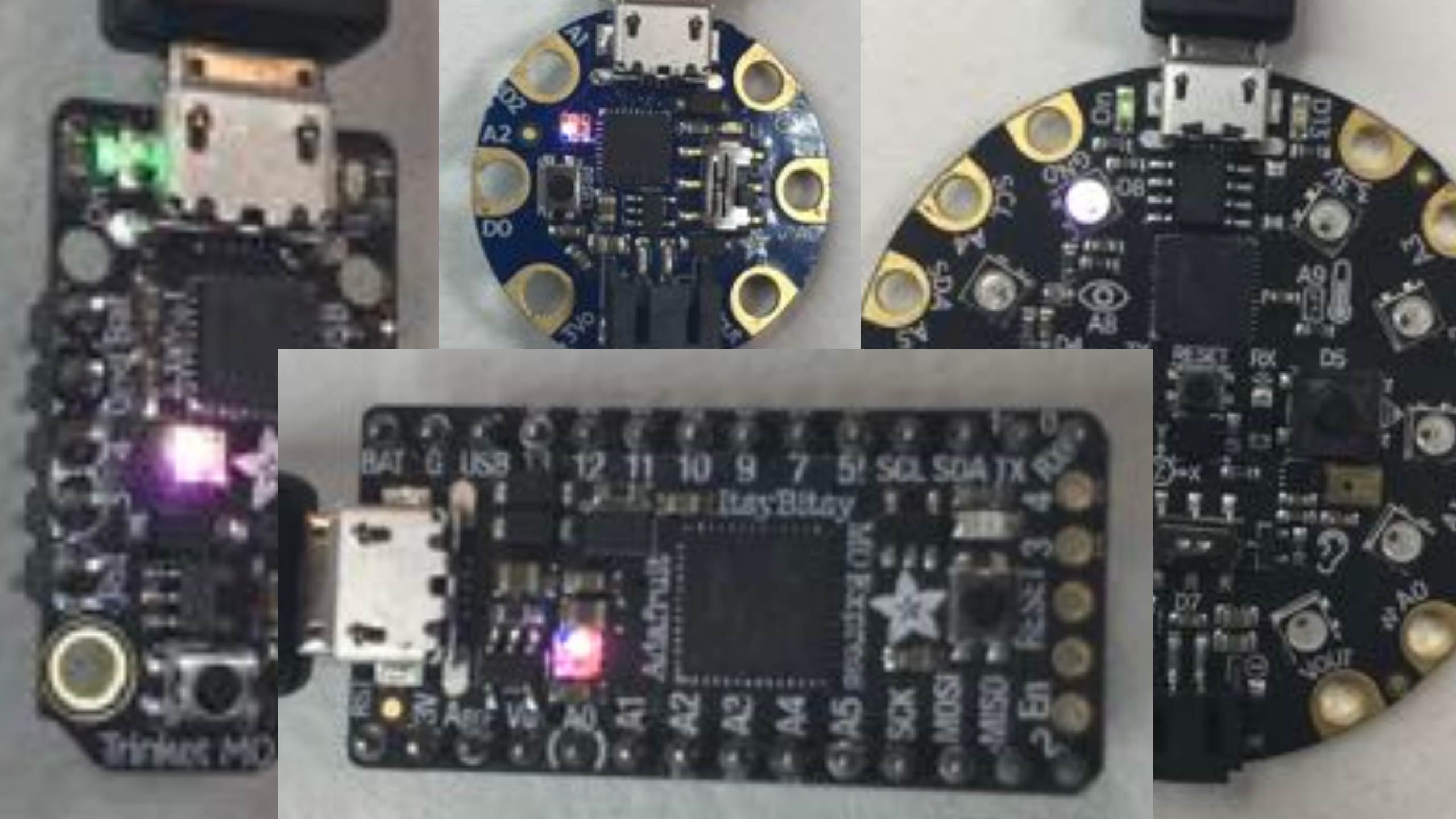
RGB LEDs: NeoPixels Vs DotStars

- Adafruit sells two primary kinds of “advanced” RGB LEDs, and they have their own names for each: NeoPixels and DotStars.
- NeoPixels are usually bigger, can be brighter, and come in RGBW styles that include a “true” white LED along with the red, green, and blue.
- DotStars connect via generic SPI (two pins), NeoPixels use a single pin and a proprietary protocol.
- NeoPixels have very specific timing requirements.
- DotStars can be more easily adopted in projects.

In Our Demo Circuits: LEDs

- All of our boards have two built-in LEDs that we'll use in our demos.
- Each has a standard red LED on pin 13.
- Each has at least one RGB LED:
  - The pin(s) vary from board to board.
  - DotStars on some boards.
  - NeoPixels on others.
  - The CircuitPlayground Express has 10!




Trinket M0
A2
D0
BAT
USB
12
11
10
9
7
5
SCL
SDA
TX
ItsyBitsy
A0
A1
A2
A3
A4
A5
SCK
MOSI
MISO
RX
D5
A0
D7

Analog Input: Resistive Sensors

Resistive Sensors: Basics

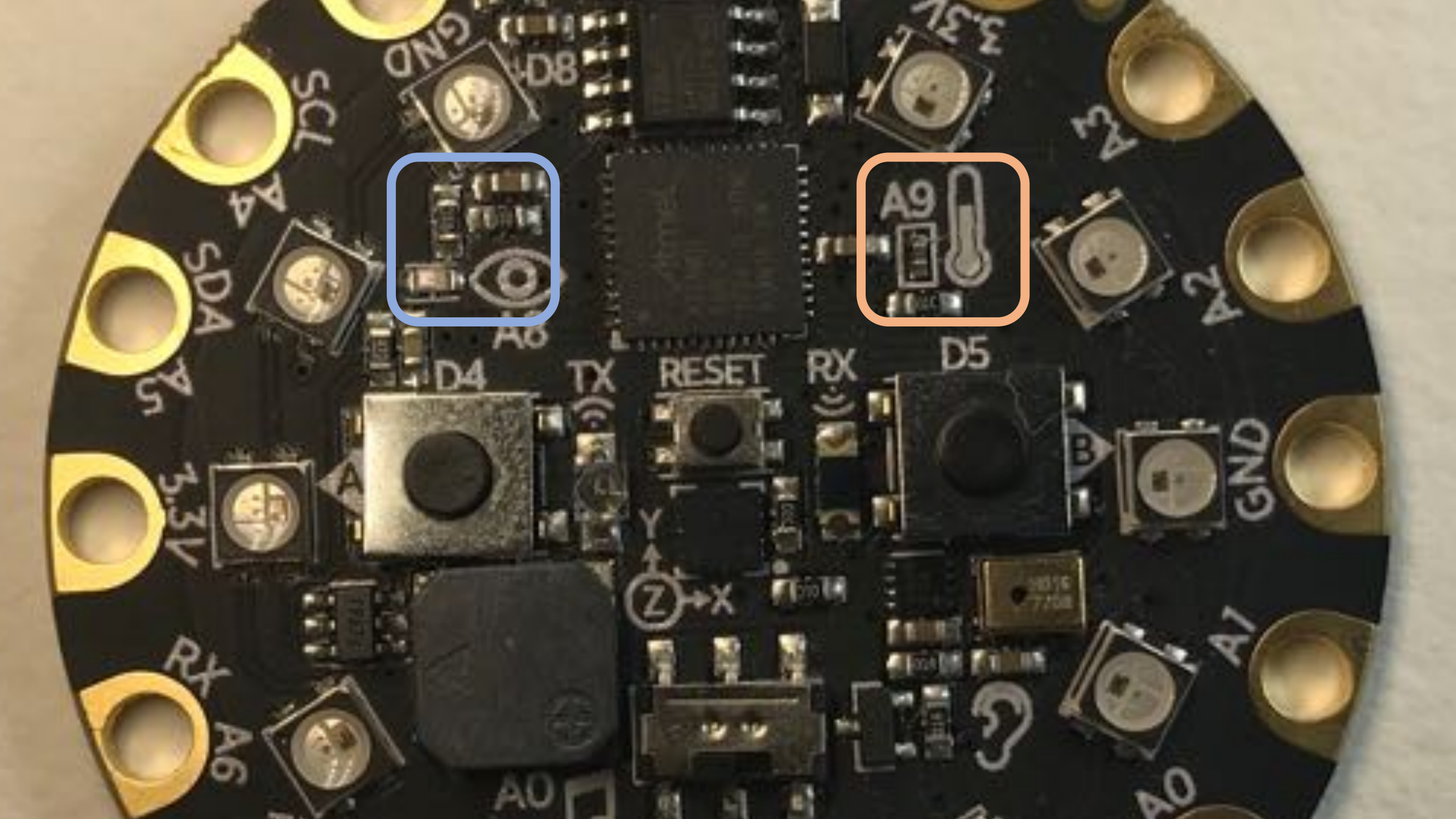
- Some metals and chemicals change their electrical resistance when they interact with other kinds of energy.
- Examples include thermistors (heat), photocells (light), and FSRs (force-sensitive resistors).
- You can use the fundamental circuit called a *voltage divider* to read the resistance with an analog input.

Voltage Dividers In Brief

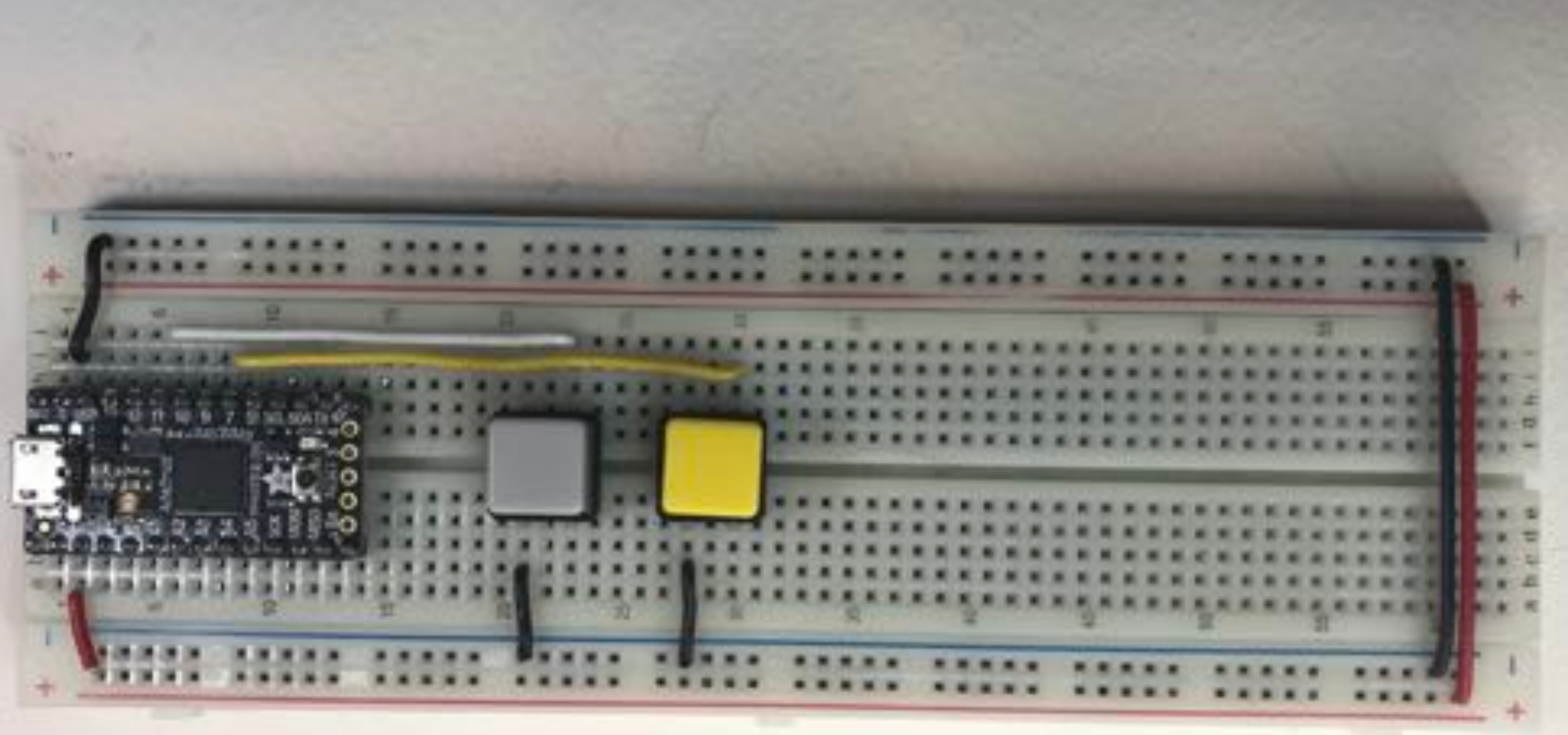
- Voltage dividers use two resistors to reduce a voltage.
- Normally, they are fixed – for example, you can use two 10kΩ resistors to reduce a 5 volt power source to ~2.5 volts.
- The amount of change depends on the values of the resistors and the input voltage, but there is a known equation to accurately calculate the expected voltage for two given resistors.
- You connect one resistor to ground, the other to positive voltage, and “tap” into the space between them to get your reduced voltage.

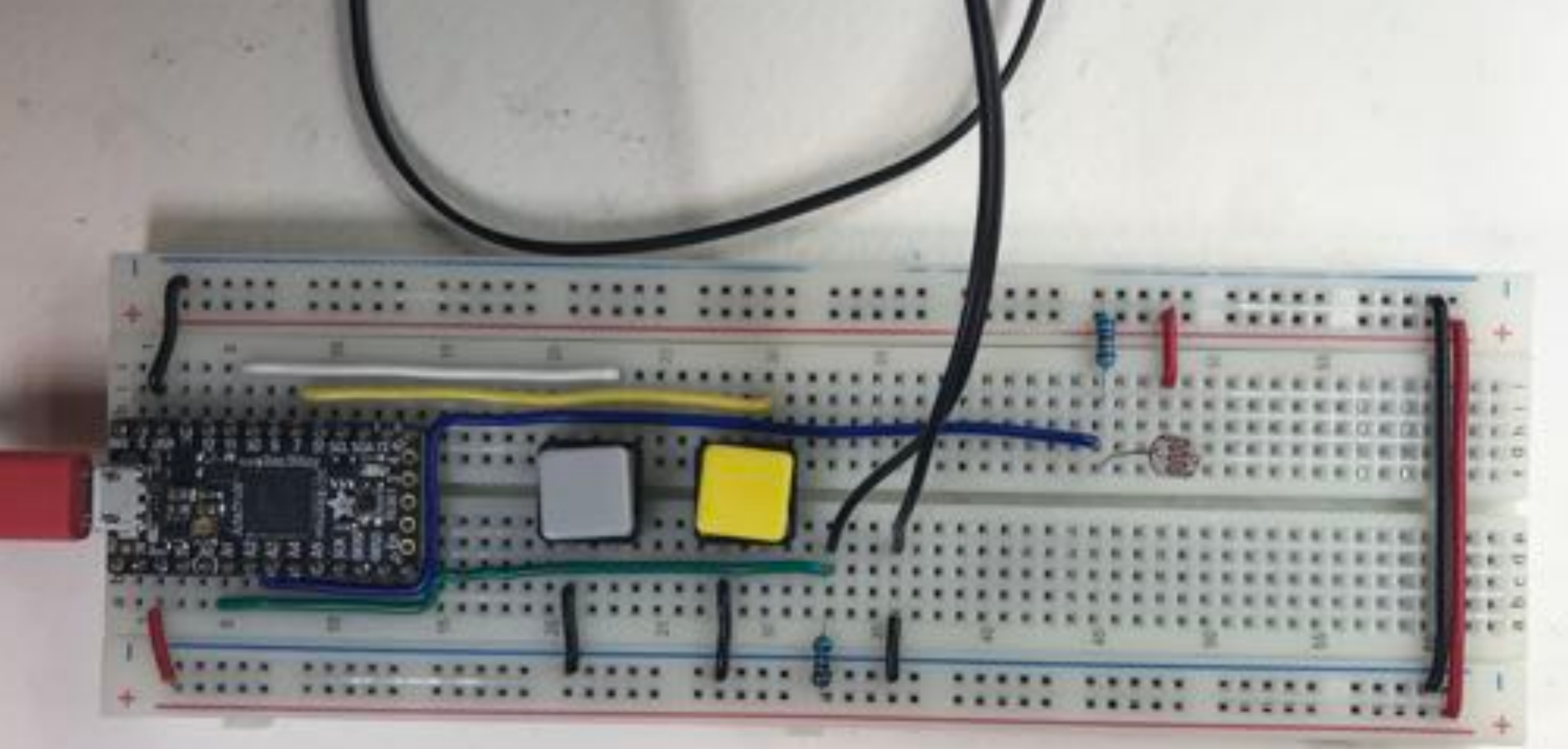
In Our Demo Circuits: Resistive Sensors

- We've wired up two common resistive sensors: a thermistor and a photocell. Both are wired along with a 10K ohm resistor.
- The CircuitPlayground Express has similar sensors built in.


GND
D8
3.3V
SCL
A4
A3
SDA
A5
A2
A9
A6
D4
TX
RESET
RX
D5
3.3V
GND
Y
Z
X
RX
A6
A1
A0
A0

The Demo Circuits





I/O In Code

Digital I/O In Code

- We use the built-in digitalio module.
- Here is the simplest possible code that will test our buttons:
  - Test for CircuitPlayground Express.
  - Test for other boards.
- Here is the simplest possible code that will test our red LED:
  - Test for all boards.

RGB LEDs In Code

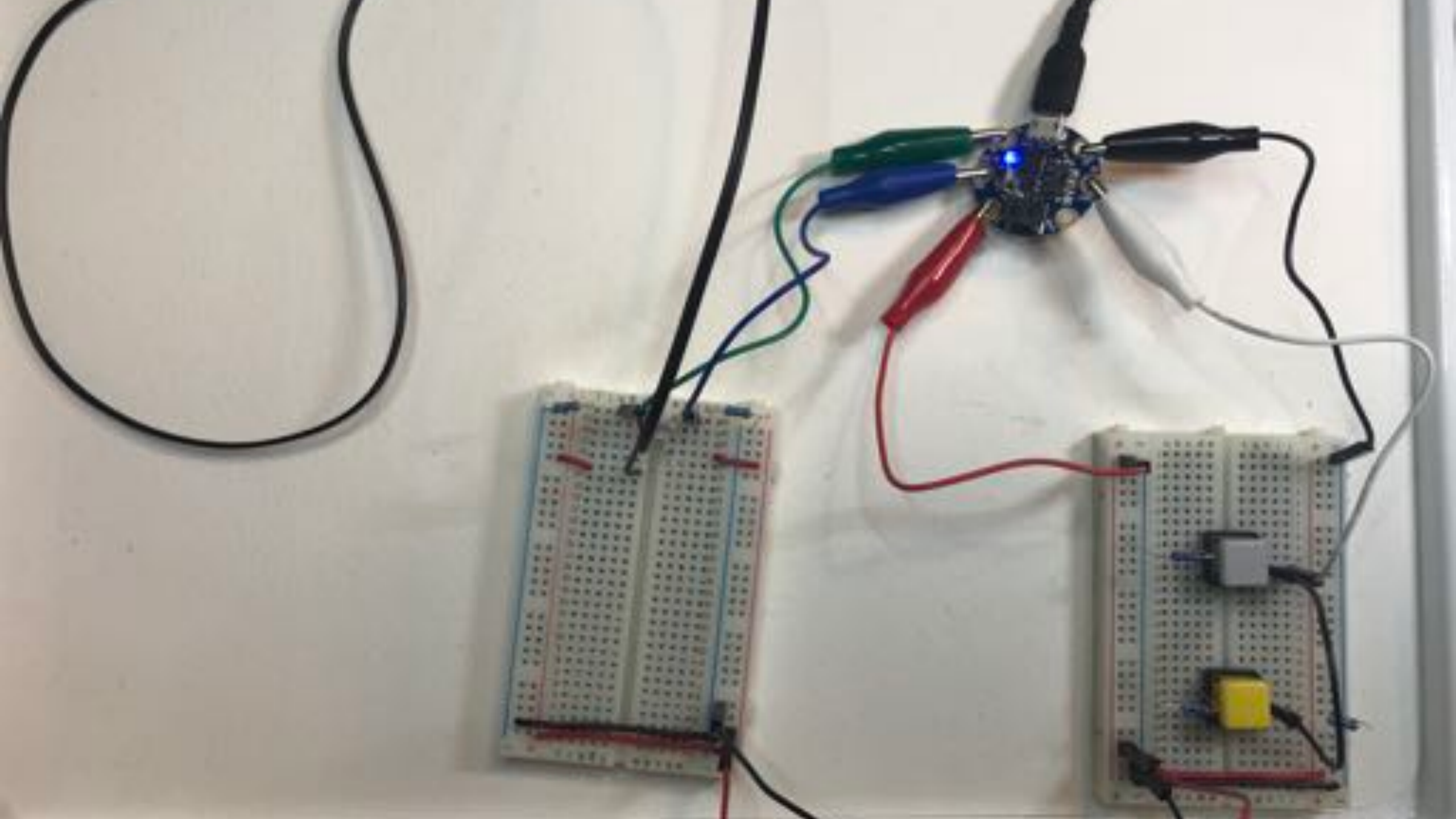
- We'll need to download and install the adafruit libraries for whichever kind of RGB LED we have on our board.
- Dotstar: adafruit_dotstar
- Neopixel: neopixel
- Here's the minimal example code:
  - NeoPixels
  - DotStars

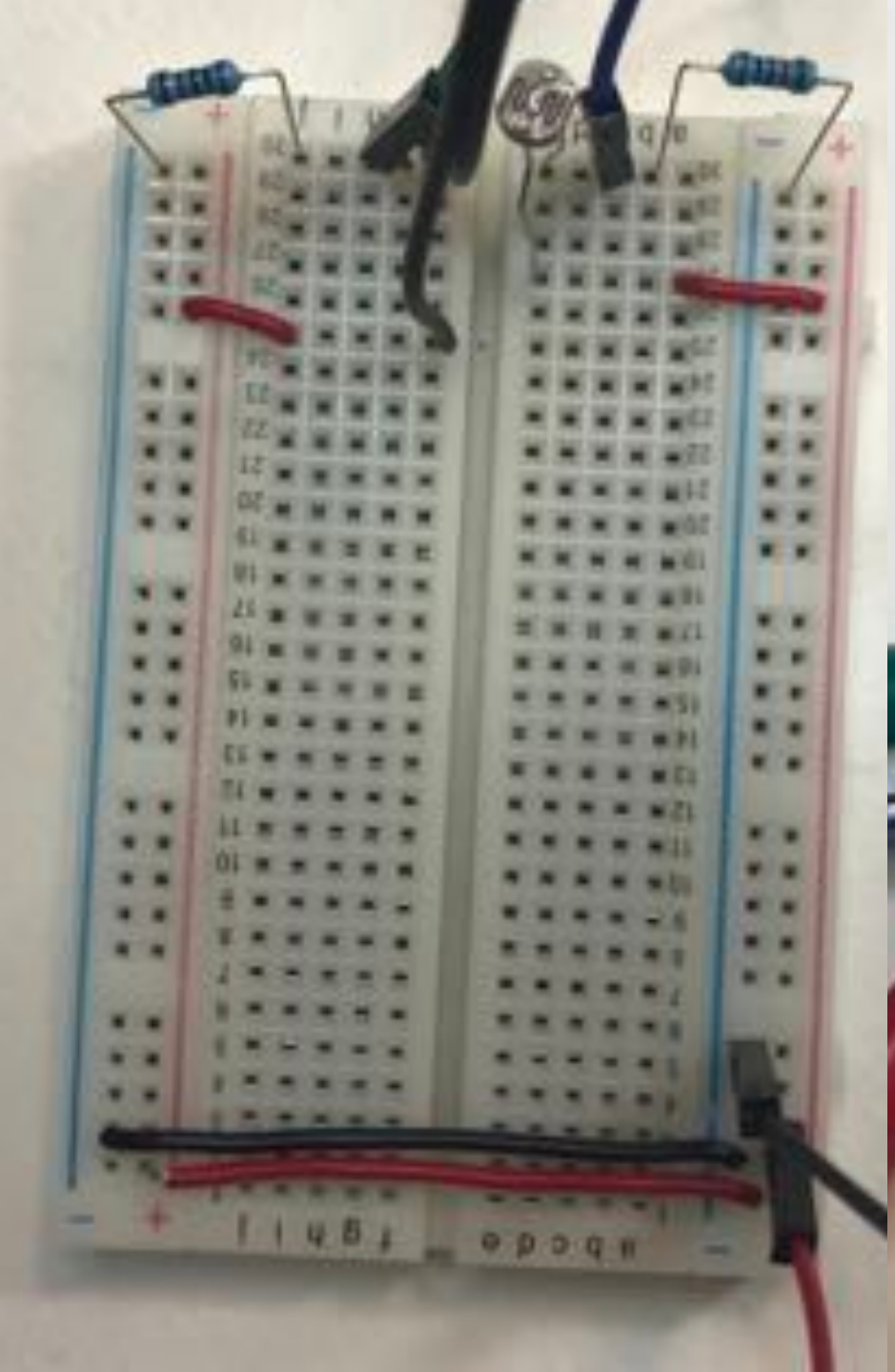
Analog I/O In Code

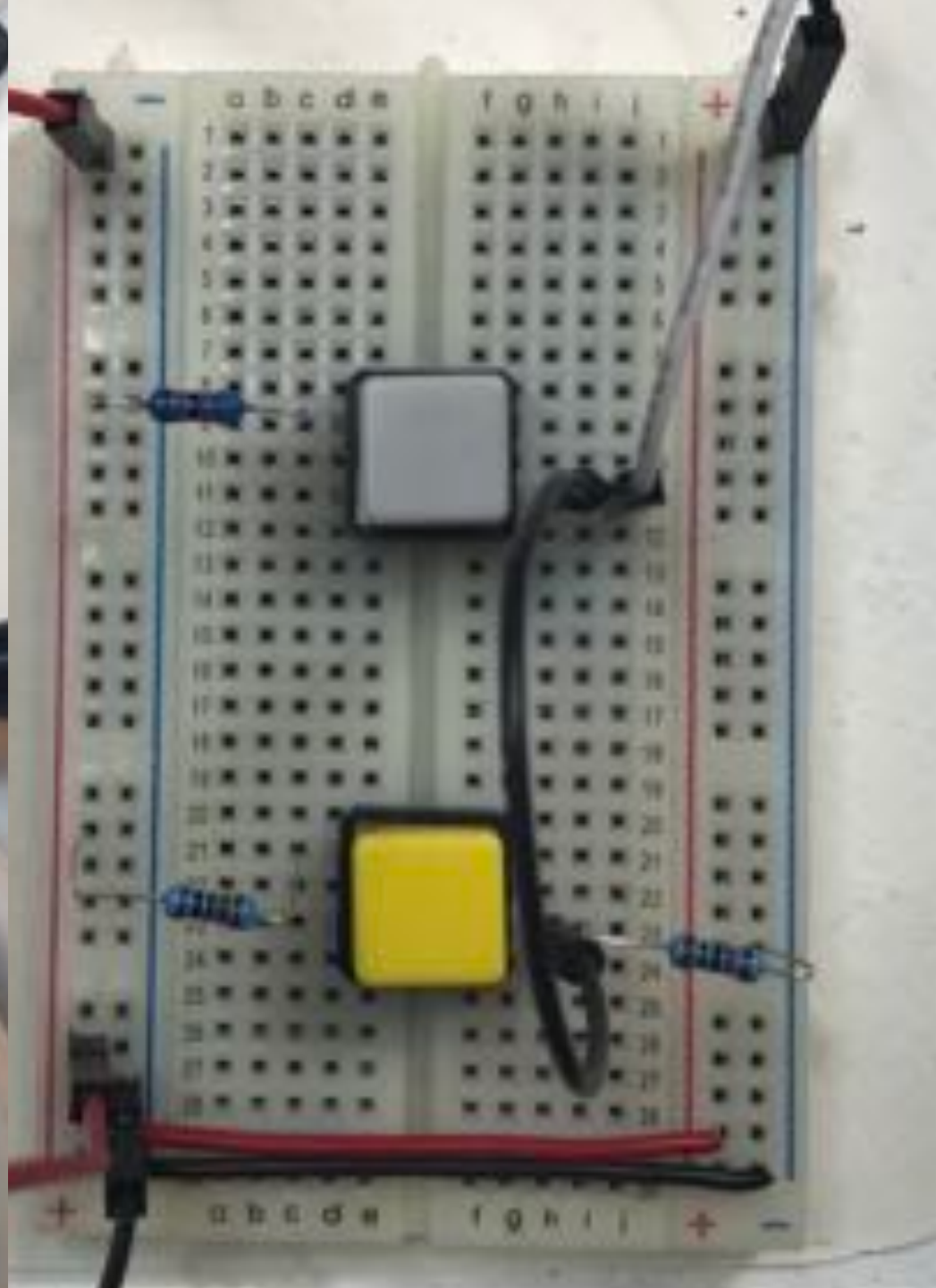
- We use the built-in `analogio` module for the photocell
- We download and install the `adafruit_thermistor` library for the thermistor.
- Here is the basic code for working with a thermistor and photocell:
  - For the CircuitPlayground Express.
  - For other boards.

Side Quest: GEMMA's Lack Of Inputs

- The GEMMA absolutely *tiny*, and that's one of the great things about it.
- However, because of this, it has a severely limited number of pins available.
- If you want to use a GEMMA with the combined digital buttons and the analog sensors, you'll have to use an old trick that takes advantage of voltage dividers to "multiplex" many digital buttons onto a single analog input.
- **TODO: retake pics with proper wiring of thermistor and photocell.**







Multiplexed Buttons: GEMMA

- Example code showing the buttons working.

Programming Philosophies

Abstraction

The Problem

- Each board provides different peripherals and GPIO.
- We want the same code to work on multiple dev boards.
- I wanted to keep my code samples as concise as possible.

The Solution

Figure out what's common, and build an application programming interface (API) to hide, or *abstract* the differences.

What's Common

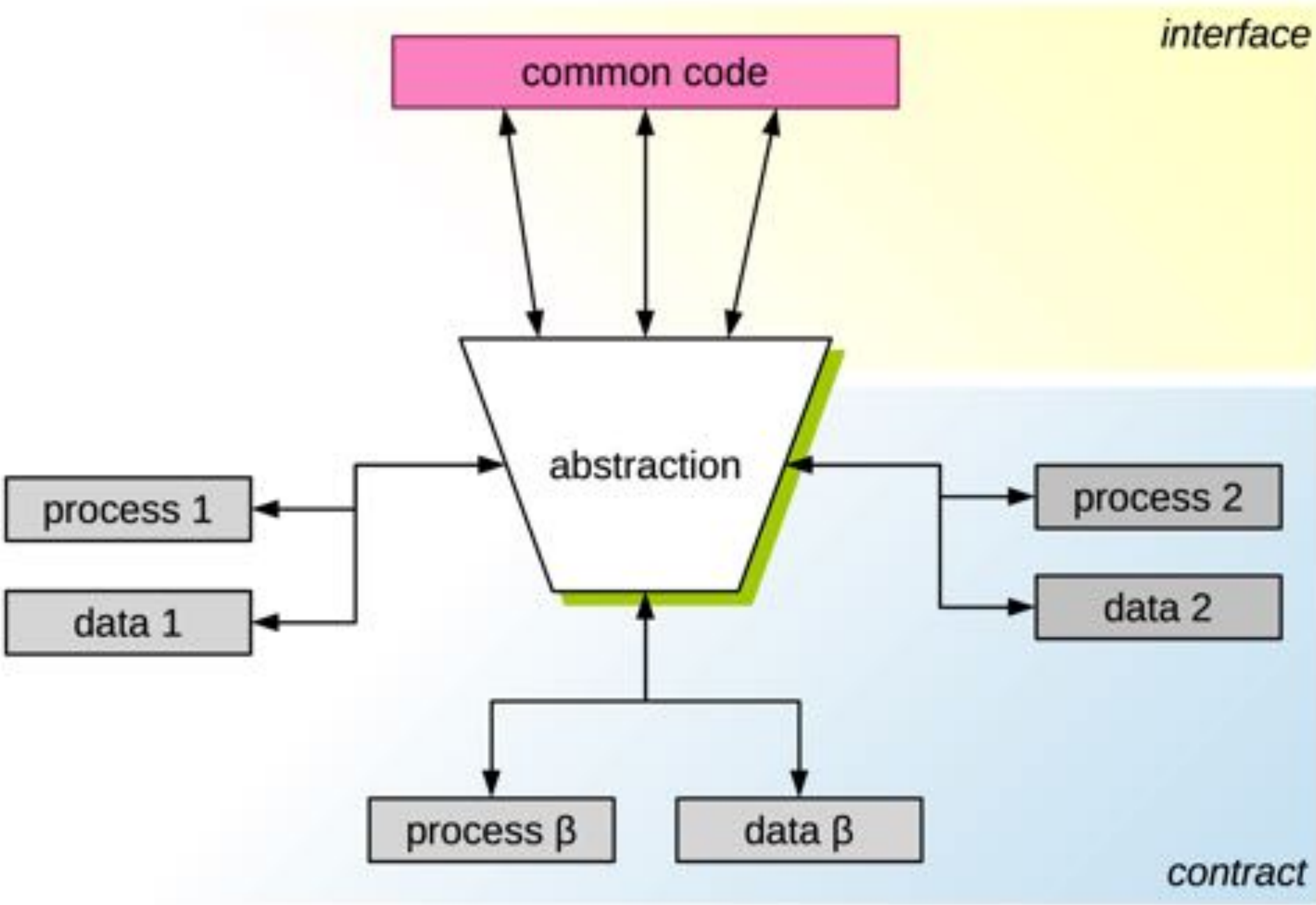
- Red LED (always on pin 13)
- RGB LED
- Button A
- Button B
- Thermistor
- Photocell

What's Different

- RGB LED could be a DotStar, could be a NeoPixel. They will required different helper libraries and could use different pins.
- The pins used for each button could be different
- If using the "multiplex" input-conservation scheme (GEMMA), the buttons are read as analog signals and compared to an expected change in voltage.
- CPX has buttons wired with pull-up resistors, other boards use pull-down – so whether a button is pressed or not will be different from board to board.
- The thermistor and photocell could be of different types. In fact, the exact type of both is slightly different on the CPX.

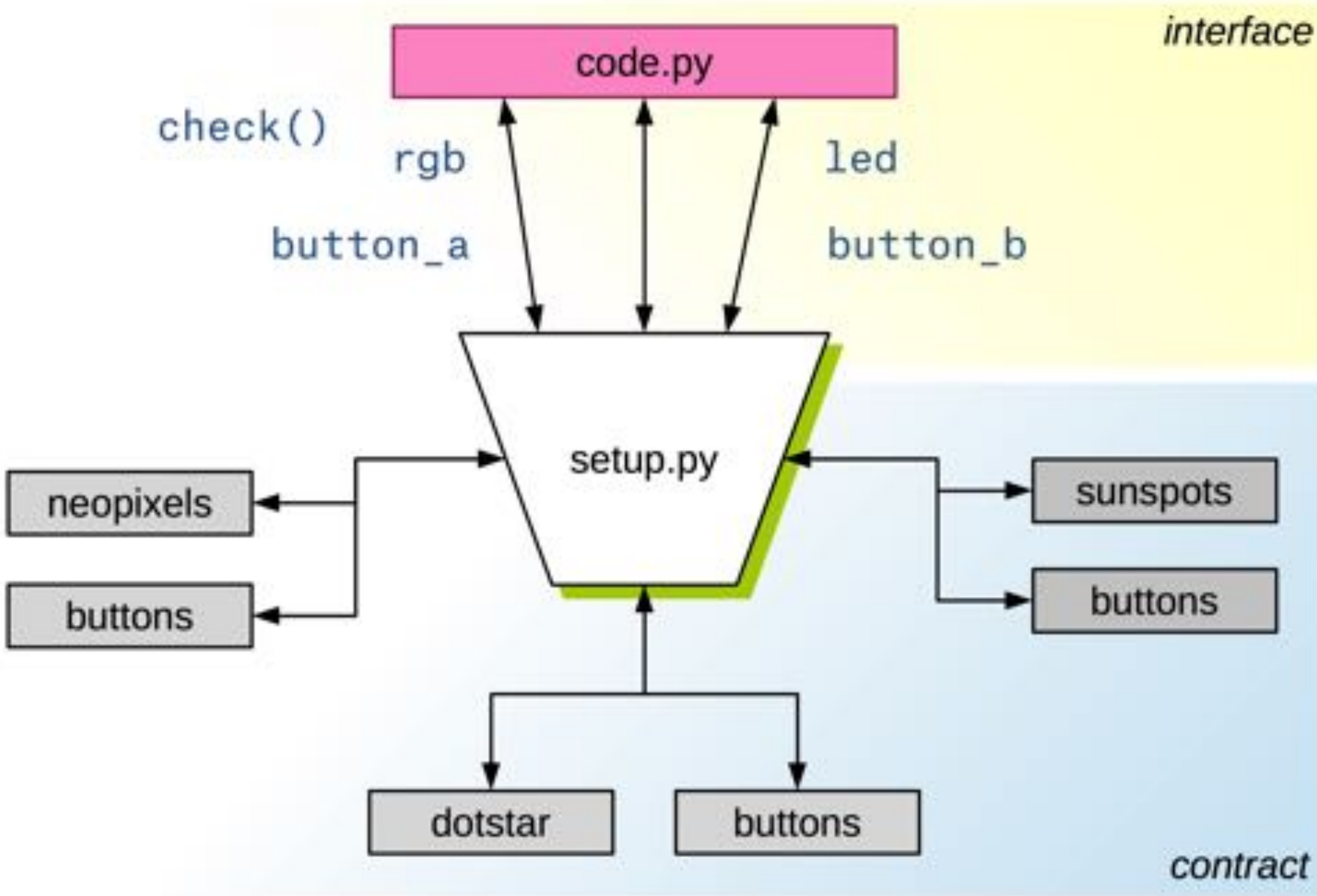
Abstraction Defined

- Abstraction is used to provide a common API to dissimilar code.
- You expect the same objects, the same methods, the same functions.
- But the implementation can vary wildly behind the abstraction.


interface
common code
abstraction
process 1
data 1
process 2
data 2
process β
data β
contract

setup.py

- A simple python *module*
- Defines our agreed upon API.


interface
code.py
check()
rgb
led
button_a
button_b
setup.py
neopixels
buttons
sunspots
buttons
dotstar
buttons
contract

Looking At The Abstraction

- setup.py for CircuitPlayground Express
- setup.py for GEMMA M0
- setup.py for ItsyBitsy M0 Express
- setup.py for Trinket M0

Using The Abstraction

- Example code that turns on the red LED when button A is pressed, and lights the RGB LED white when button B is pressed.

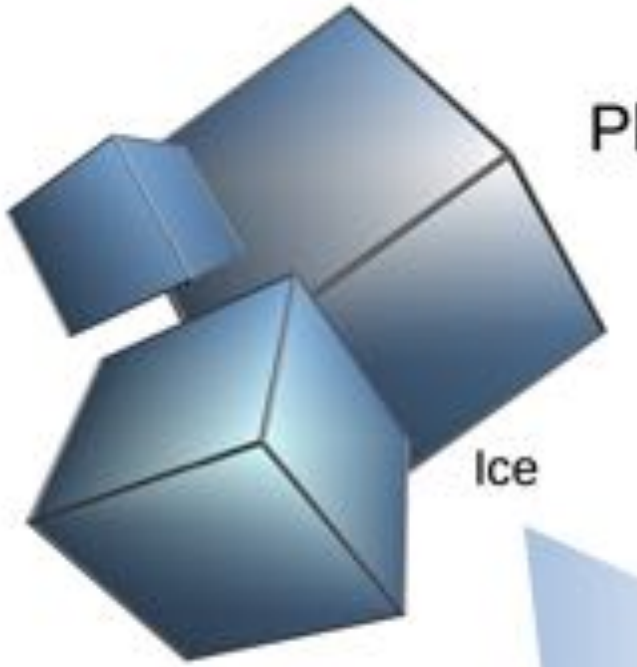
Basics Of State

What Is State?

- *State* is the status of a thing, a collection of properties.
- State describes what something looks like at a given moment in time.
- *State* can *transition.* It changes.
- *State* can be the properties of your entire project, or just one part of it.

Phases (**States**) Of Water

and some transitions



Ice

Steam

water freezes

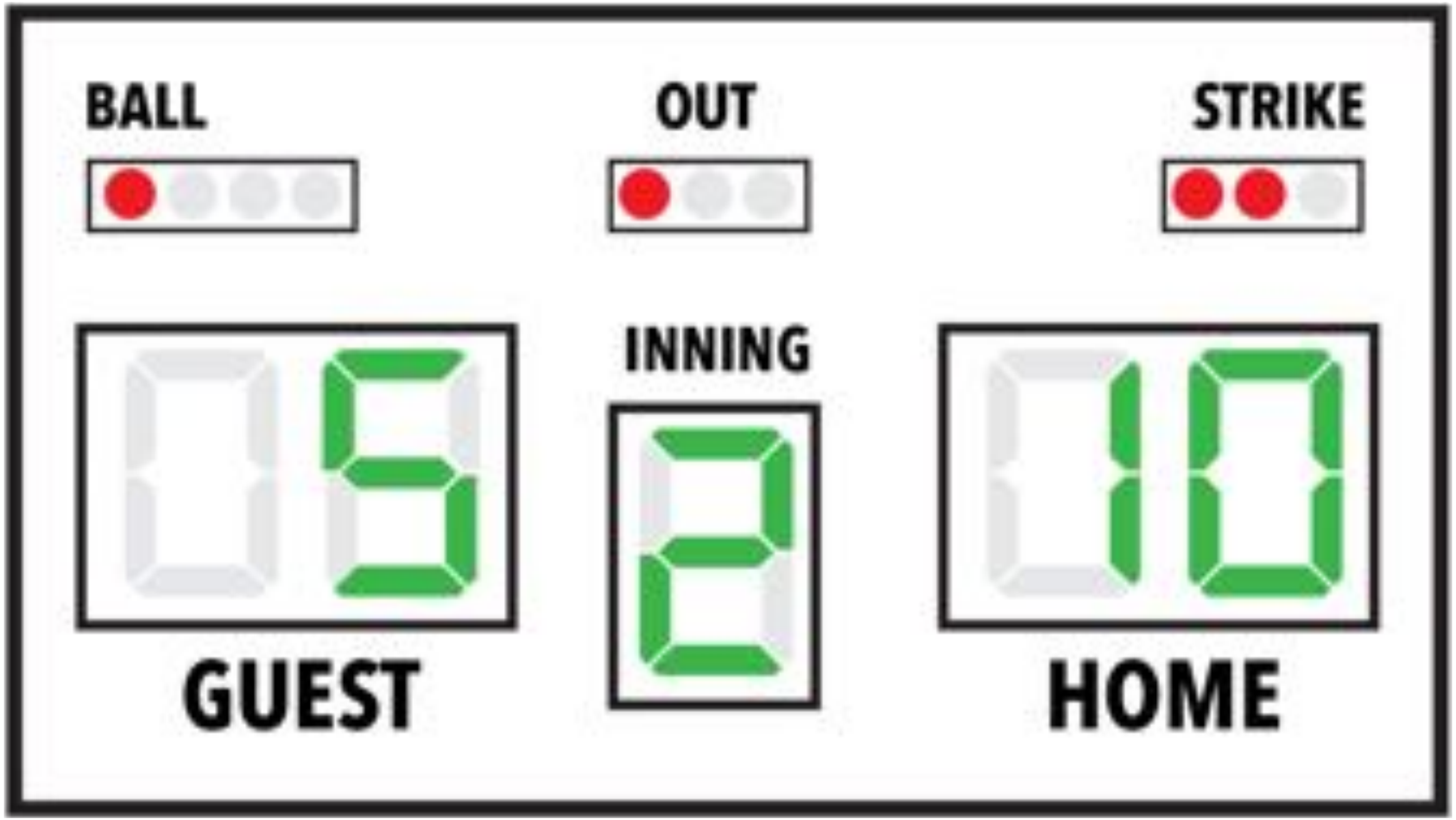
steam condenses

ice melts

water boils

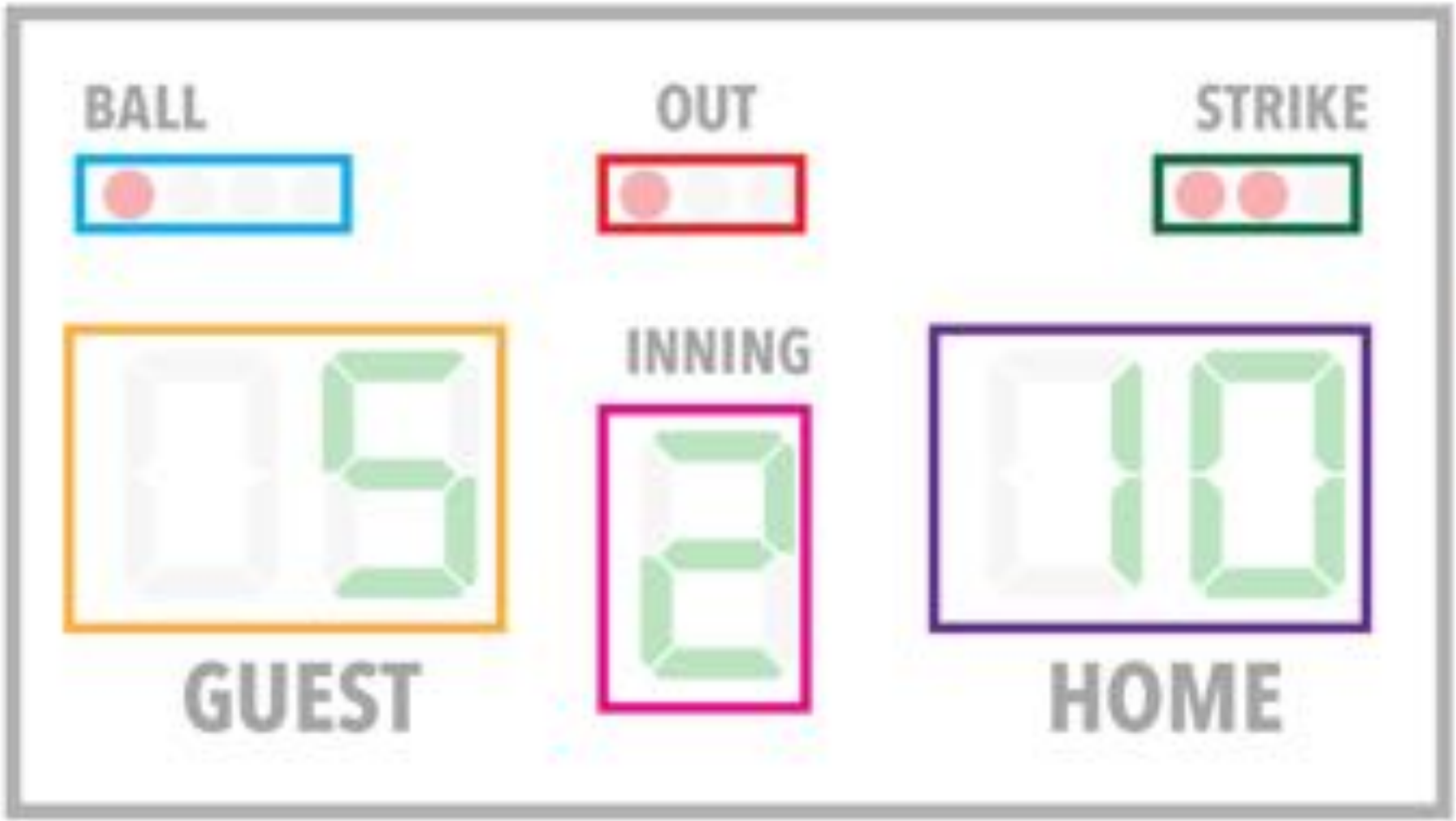
Water

The Scoreboard Analogy


BALL
OUT
STRIKE
INNING
GUEST
HOME

The Scoreboard

- Represents *global* state.
- Changes over time.
- Changes when things happen in the game.




```
ball = 2
out = 1
strike = 1
guest_score = 2
inning = 5
home_score = 10
```

State Can Be Stored As Any Mutable Object

- Simple Variables
- Lists
- Dictionaries
- Strings
- Bitfields
- Any combination

Mutable Vs Immutable

Mutable objects can be changed.

Immutable objects cannot.

Scoreboard Implemented

- Simple variables
- List
- Dictionary
- Combination

Another (Better) Way: Classes

Classes

- Classes arrange data into a given structure. They *model* data.
- Classes are a “Blueprint” for making new objects that contain similar data.
- Classes are used to create *instances* – objects that contain a copy of the data structure.
- Classes can contain *methods*, or functions that operate on instances or classes.

Instances Vs Classes

- **Classes** are blueprints used to make instances and hold separate class-specific data.
- **Instances** are copies of the class and hold their own data.

Class Vs Instance *Data*

- Class data exists *once.*
- Instance data exists many many times (0 to ∞ times).

Contrived Example

- A simple class, making instances, methods explored, instance data vs class data.

The Scoreboard As A Class

- Here, we've implemented a (somewhat contrived) model of an American baseball scoreboard.

What's Great About Classes

- Classes *encapsulate* data in an intuitive way.
- Classes reduce the amount of code you have to write through the use of *polymorphism* (more on that shortly)
- Classes have some really cool features that are hard to code otherwise, for example, you can make methods that act like properties.
- Classes in Python have so-called "magic methods" so it's easy to make them work like built-in types.

Why *not* use classes?

- Classes can be more difficult to debug.
- Classes don't always use less code, sometimes they force you to write more.
- The biggest cases against them, however, is because this is CircuitPython and *every line of code matters*:
  - *every single line of code* has to fit within available RAM.
  - we're limited in terms of storage space for code.
- Classes generally take more lines of code to write and use more RAM than built-in datatypes.

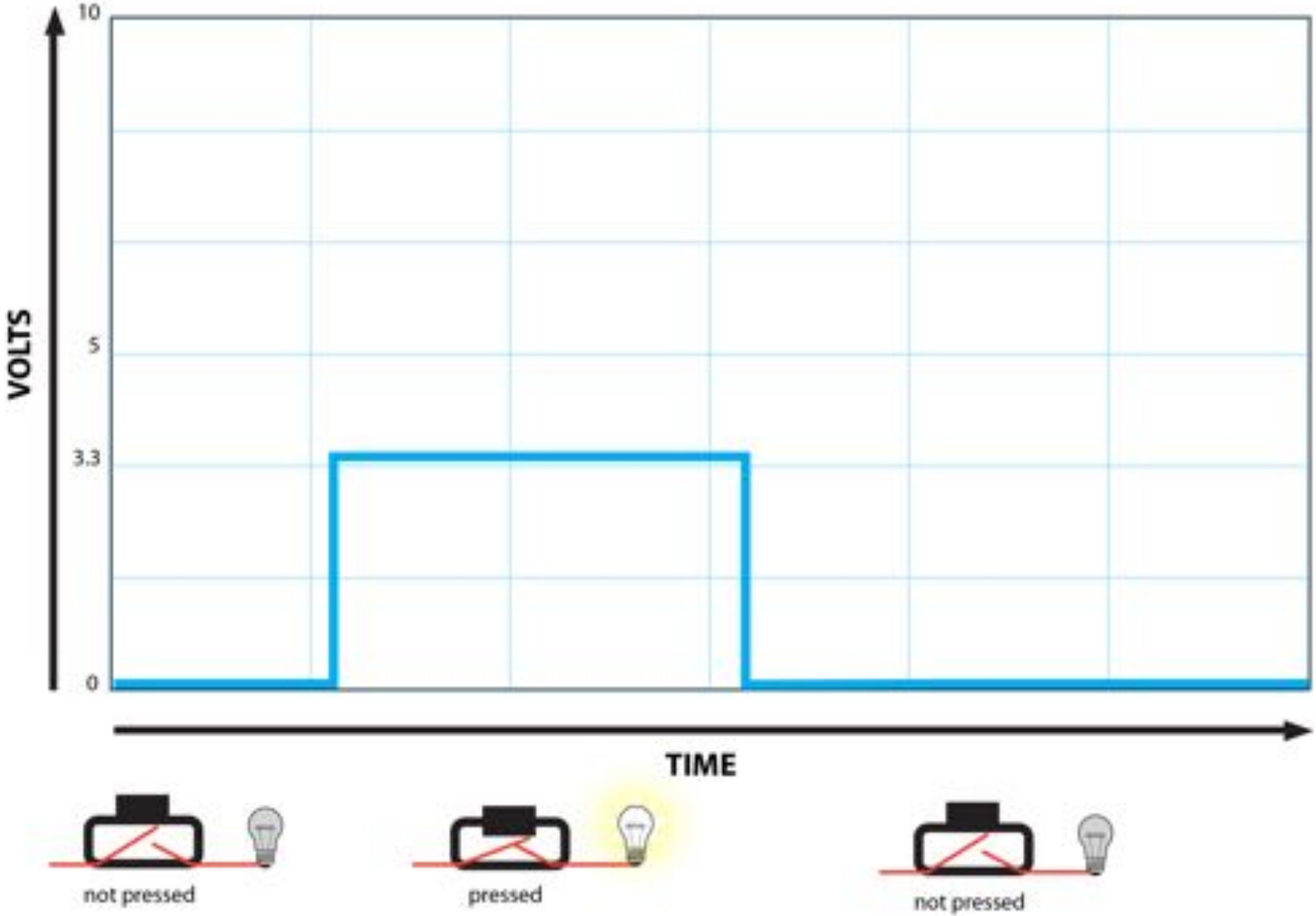
Button State

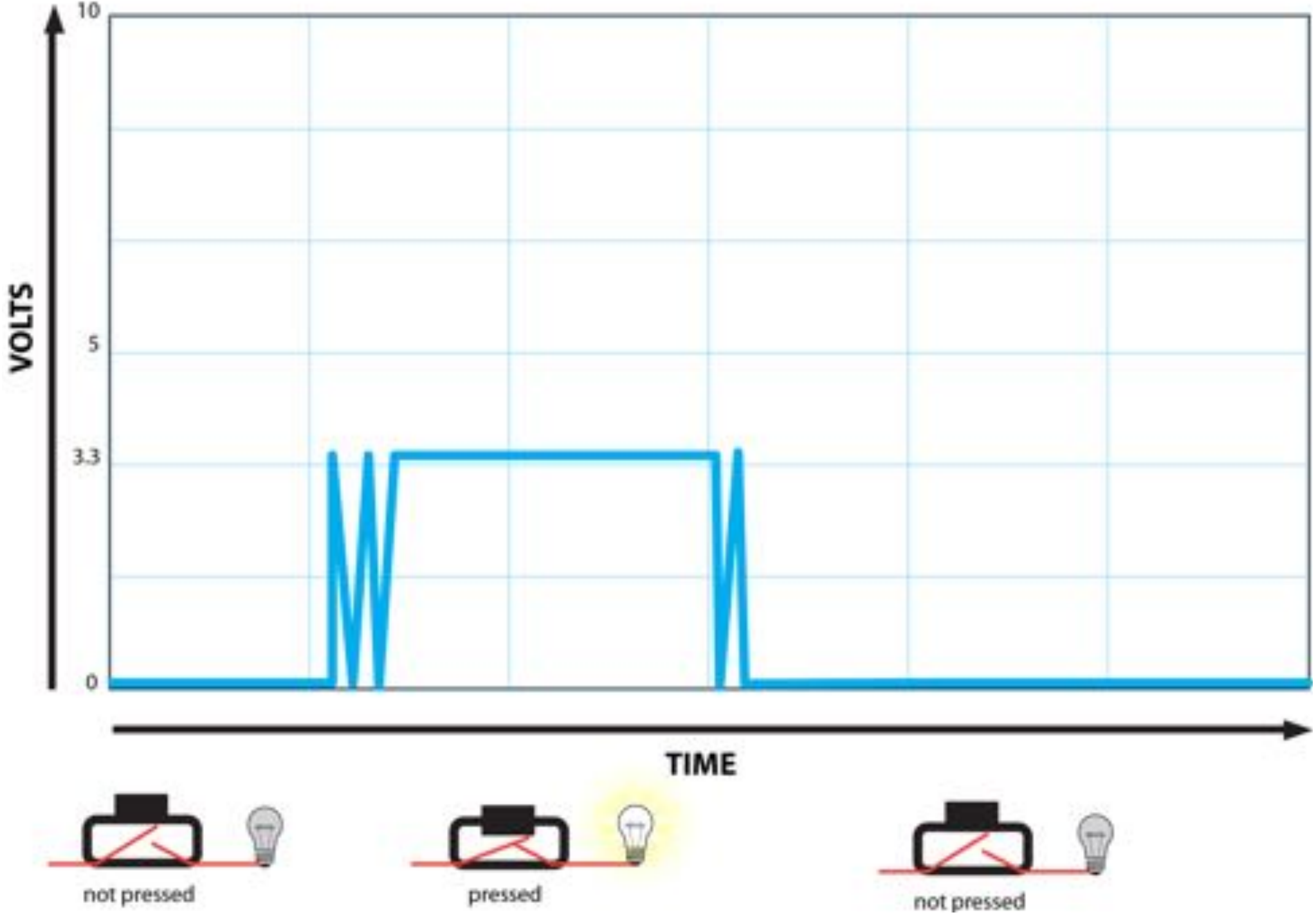
Momentary Buttons And State

- Buttons have inherent state – they are ”on” or “off”.
- We can use our state modeling concepts to track button state.
- We can extend the basic state tracking to do cool things like detect events.
- But most critically, state is key to fighting the dreaded **button bounce.**

Button Bounce Explained

- Buttons are not perfect.
- People are not perfect.
- MCUs are not perfect either.
  - Recall there is a range in play for detecting whether a button is on or off.
- Electricity wants to flow, so it will flow.
- In a main loop, we're checking button state *millions* of times per second.
- These facts conspire to give us "noisy" signals from buttons – this is called "bounce".


10
5
3.3
0
VOLTS
TIME
not pressed
pressed
not pressed


10
5
3.3
0
VOLTS
TIME
not pressed
pressed
not pressed

De-bouncing: A Sampling Problem

- The issue is, at its heart, an issue of *sampling*.
- Every loop, we're sampling the value of the button, and taking action.
- Electricity moves near the speed of light, our code runs millions of times per second.
- We are getting as close to real-time status as possible, this is why we get the noise in our signal.
- To "smooth out" the data, we just need to **reduce the sample frequency**.

How Might We Reduce Sample Frequency?

- We can slow down the button (use an R/C circuit)
- We can slow down the processor.
- We can check less often.

Why Not An R/C Circuit

- R/C circuits use a “network” of resistors and a capacitor to store up a charge over time, and then discharge it all at once.
- It requires more components, so it’s more expensive and (more importantly for hobbyists) it’s more error-prone.
- It’s out of the scope of this talk (it’s worthy of an entire talk on its own, R/C circuits are one of the most versatile circuits in electronics).
- I’ve never gotten it to work. 😔

The Worst Best Way: Slowing Down The MCU

- The easiest way to slow down the processor is to *block*
- Blocking means tying up the processor so it can't execute any other statements.
- In Python, the typical way to do this is with `time.sleep()`
- We've been doing this in our testing code.

Why Not Block?

- When the processor is blocked, nothing can happen – no inputs can be read, no outputs altered, no variables can be manipulated.
- We become tied to a single fixed frequency.
- Blocking is perfectly acceptable for simple projects.
- However, as things get more complex, blocking becomes a problem.

State To The Rescue!

- We can use state to track the passage of time.
- We start by storing the current time in a state variable.
- Every loop, we check to see how much time has passed.
- If enough time has passed to properly reduce our sample frequency, we take action.

Simple State To Reduce Sample Frequency

- Example code

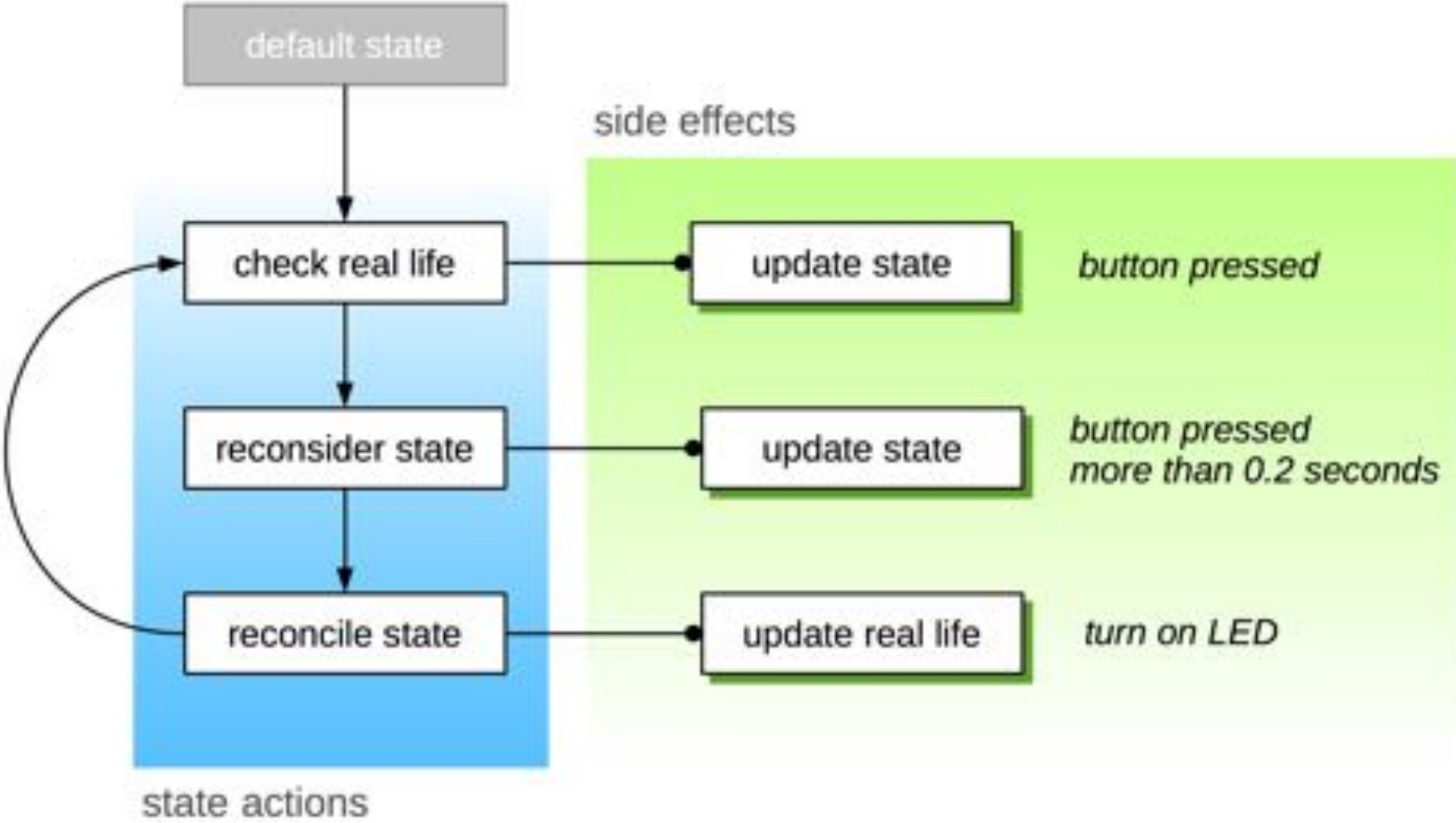
Using A Class To Do The Same Thing

- Example code

Three-Pass State Management

Consider the following example code

- Example of using state to control the color and status of the RGB LED.


default state
side effects
check real life
update state
button pressed
reconsider state
update state
button pressed
more than 0.2 seconds
reconcile state
update real life
turn on LED
state actions

Phases Explained

- Default State

  The initial state when the board boots up
- Check Real Life

  Read the physical state of buttons, sensors, etc.
- Reconsider State

  Look at the state as a whole, and change the state based on what you see.
- Reconcile State

  Interact with the physical world, turn on the LED, take other action.

Why?

- Keeping state and the physical sensors/LEDs apart is a fundamental *separation of concerns*.
- Don’t think of each line of code as executing in sequence – things are happening so fast that the lines are effectively **interleaved**.
- Sensors and LED libraries can necessarily block – we want to get out of the way quickly and let them do their thing.
- This way of looking at things opens the door to some really cool stuff, in particular, *event detection*.

Event Detection

What is an event?

- An event is a moment in time that matters to your application.
- Most of the time, events happen when state transitions from one value to another.
- But it can also be arbitrary - for example, when our 'debounce' time has elapsed, that is an event.
- Events can also happen because of other events, or because state changes in a certain way – for example, if you hold a button down for 5 seconds. That's the debounce event, then "the button was pressed" event, followed by "five seconds has elapsed".

Button Events

Button Events

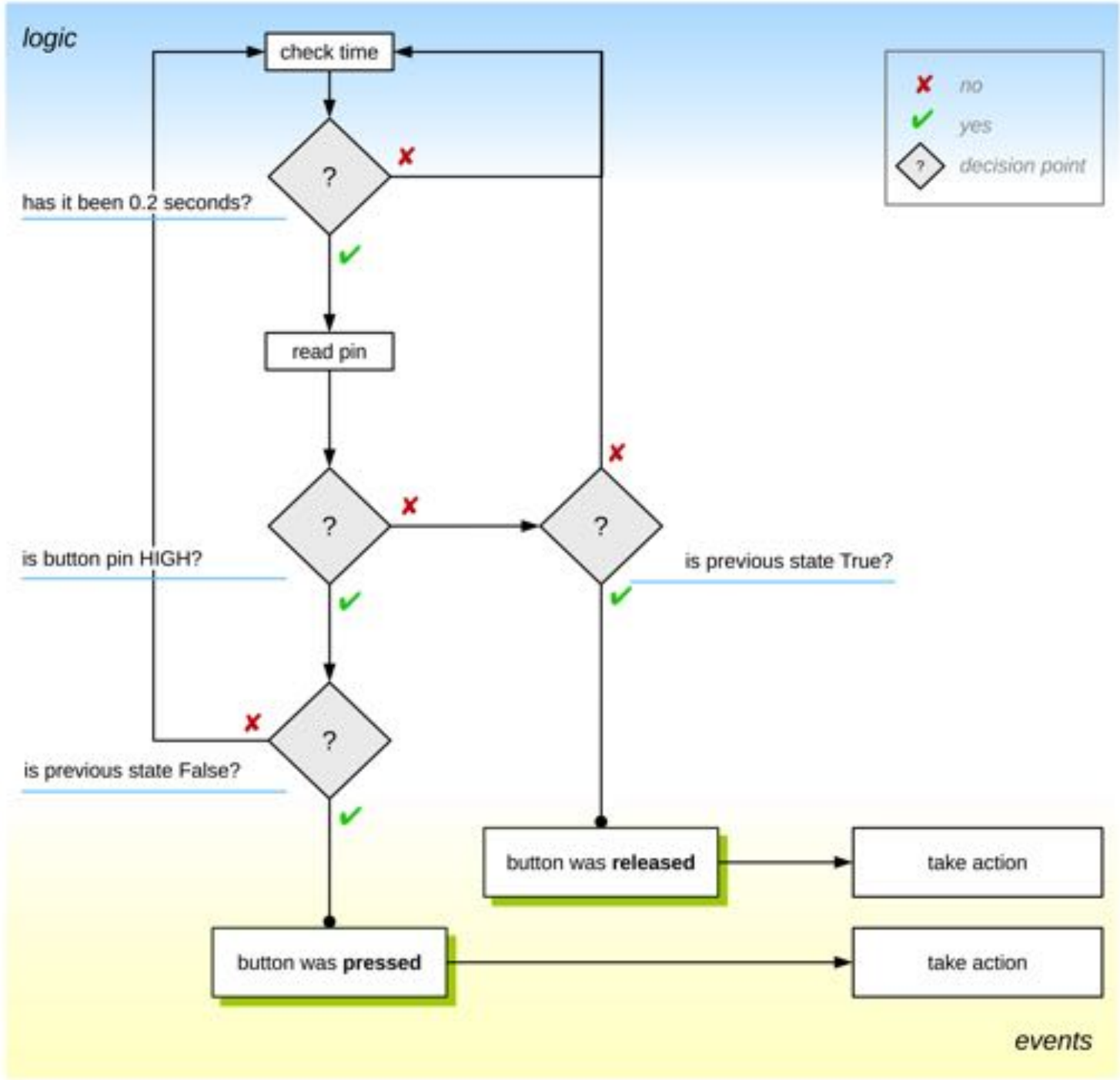
- Buttons have several possible events as their state changes when people interact with it.
- Lets start with the two primary ones: **press** and **release**
- These primary events can be defined using some simple logic.

Press Event

- The debounce time has elapsed.
- The state of the button was False, and the button is now reading True.

Release Event

- The debounce time has elapsed.
- The state of the button was True, and the button is reading False


logic
check time
?
has it been 0.2 seconds?
read pin
?
is button pin HIGH?
?
is previous state True?
?
is previous state False?
button was released
take action
button was pressed
take action
events
no
yes
decision point

Button Events In Code: Basics

- Example of basic button event detection

Button Events In Code: A Simple Class

- Example of using simple fixed classes to do button events

Flexible Event Detection: Polymorphism

- Our previous example was fixed, all of the code is contained within the two very similar classes.
- The event detection and debounce code will be the same regardless of the use case of our class.
- We can factor the common code into a *base class* and use *inheritance* to re-use that code in new classes.
- We can write new code in the derived, or child classes to handle the events in different ways.

A Contrived Example

- Example of how class inheritance works.

Button Event Detection: Polymorphic

- Example of a base class for button press and release detection, and classes that use the base class to do special things with the events.

Analog Event Detection

- Detecting events that happen with our analog sensors is similar to how we detect events with buttons.
- However, we have a wide range of values instead of just True/False.
- Analog sensors are also much more *fuzzy* than their digital cohorts.
- This is especially true on our chosen platform: the M0/M4 processors have very sensitive ADCs, and produce values ranging from 0-65535.

Analog Events: Considerations

- Because of the fuzziness, we can't rely on an exact amount.
- We'll need to use some math, and some special sampling techniques to get a good reading.
- We're also going to want to deal with *thresholds*, as opposed to absolute values.
- *Thrashing*, or quickly changing from one state to the next is very possible and we'll need to protect against it.

Reliable Analog Sensor Reading

- Before we can detect events, we'll need to first do some sampling.
- This is very similar to how we debounced buttons earlier, except that we're going to compare values *and* check periodically.
- The goal is to get a reading that is accurate over a short period of time.

Example: Photocell Reading

- Example of noisy photocell code.
- Example of sampling the photocell and smoothing the value out.

Basic Analog Events: Low/Medium/High

- Since our examples are kind of contrived, we need to come up with some events, that are also kind of contrived.
- Let's trigger three events:
  - 'low', when the sensor reads below a certain amount
  - 'high', when the sensor reads above a certain amount
  - 'medium', when the sensor is in between "high" and "low"

Example: Temperature Events

- Example of the temperature sensor triggering the low/high/medium events.

A New Event: Change

- Our last example printed to the console *every* time the temperature sensor was read.
- This means the same “medium” event code will run over and over as long as the temperature sensor is reading a “medium” value.
- There are times when we would want the code to run *once*, when the temperature transitions from one range to the next.
- This means we have a new fourth event: ‘change’.

Example: Temperature Events And Change

- Example code showing the change event along side the other temperature range events.

Polymorphism Can Help

- If we were to implement the same code for the photocell, most of it would look very similar.
- Only the values themselves, and what constitutes “high”, “medium” and ”low” would change.
- We can use polymorphism to easily create a base class containing the common code.
- Then we can create projects using all sorts of similar sensors.
- We can also use the “change” event for unexpected use cases.

Example: A Generic Analog State Base Class

- Example of implementing code using a base class.

Implementation: Color Changing

- Example showing the minimal implementations showed in the last example being used to change the colors of the RGB LED depending on the analog range.

It’s All Great Except The Boilerplate

- The downside of our class-based event dispatchers is that they require extra classes to be written every time you want to add new functionality.
- This is often referred to as *boilerplate code*.
- Boilerplate is fine in simple cases, but as things get more complex, it can get tedious.
- As mentioned earlier, we also have to be conscious about memory and storage usage – more classes means more lines of code.

Avoiding Boilerplate: Instance Configuration

- The simplest way to avoid this class inheritance stuff is to take advantage of the *constructor's* purpose as the "setup" routine for every new instance of a class.
- Up until now, we've only used constructors for setting up our default values.
- Constructors can accept arguments like any other function or method.
- We can pass in parameters to the constructor to configure the instance and avoid having to write more boilerplate.

The Generic Analog Event Dispatcher Mark 2

- Example of the analog event dispatcher refactored to work with configuration in the constructor, and implementations of the RGB LED color changing code for the thermistor and the photocell.

Could We Go Further?

- The last example works, but it only has one function: it changes the color of the LED.
- We have to implement, at a minimum (boilerplate!!) one method: `sample()`
- Everything else is configurable in the constructor, except the actual code that is executed when an event happens.
- It's possible to take this a step further, and utilize a neat concept called *dependency injection.*

Dependency Injection

A Brief Introduction To Dependency Injection

- Dependency Injection is a pattern by which you pass *services* to methods or functions instead of creating them in the method or function.
- It's useful because you can rely heavily on abstraction: anything that has the right API will work, and how it's implemented is of no concern to the code using it.

Dependency Injection In Python

- In Python, mutable objects are passed *by reference.*
- Everything in python is an object, and only a few are immutable.
- This means we can pass objects, functions, and even instance methods as parameters.

Dependency Injection In Action

- TODO: Example of a generic button dispatcher.
- TODO: Example of passing the state object too
- Example of our configurable generic analog event dispatcher, but fully and utterly configurable using dependency injection.

END!

Applications

Nightlight

- This application turns the NeoPixel or DotStar on when the light falls below a certain threshold.
- It's configurable using the two buttons:
  - Button A cycles the RGB LED color
  - Button B cycles the intensity
  - Holding Button A toggles the LED(s) on/off.
  - Holding Button B toggles flashing.
- Example code

Touchmouse

- Low-impact mouse buttons using capacitive touch technology.
- 4 functions:
  - Left click
  - Right click
  - Scroll up
  - Scroll down
- The RGB LED changes color to give a visual indication of what's happening.

Conference Session Timer

- A visual que that tells a speaker how long they have left in a non-distracting way.
- Changes the RGB LED:
  - from white to green when the talk begins
  - from green to yellow when the talk has 5 minutes left
  - from yellow to red when the talk is supposed to be over.
  - flashes white and red when the talk has gone over by 5 minutes.
- The A button is used to pause or start the timer. Hold for reset.
- The B button asks for more time. If you have surpassed your limit for extra time requests, the LED briefly flashes blue.