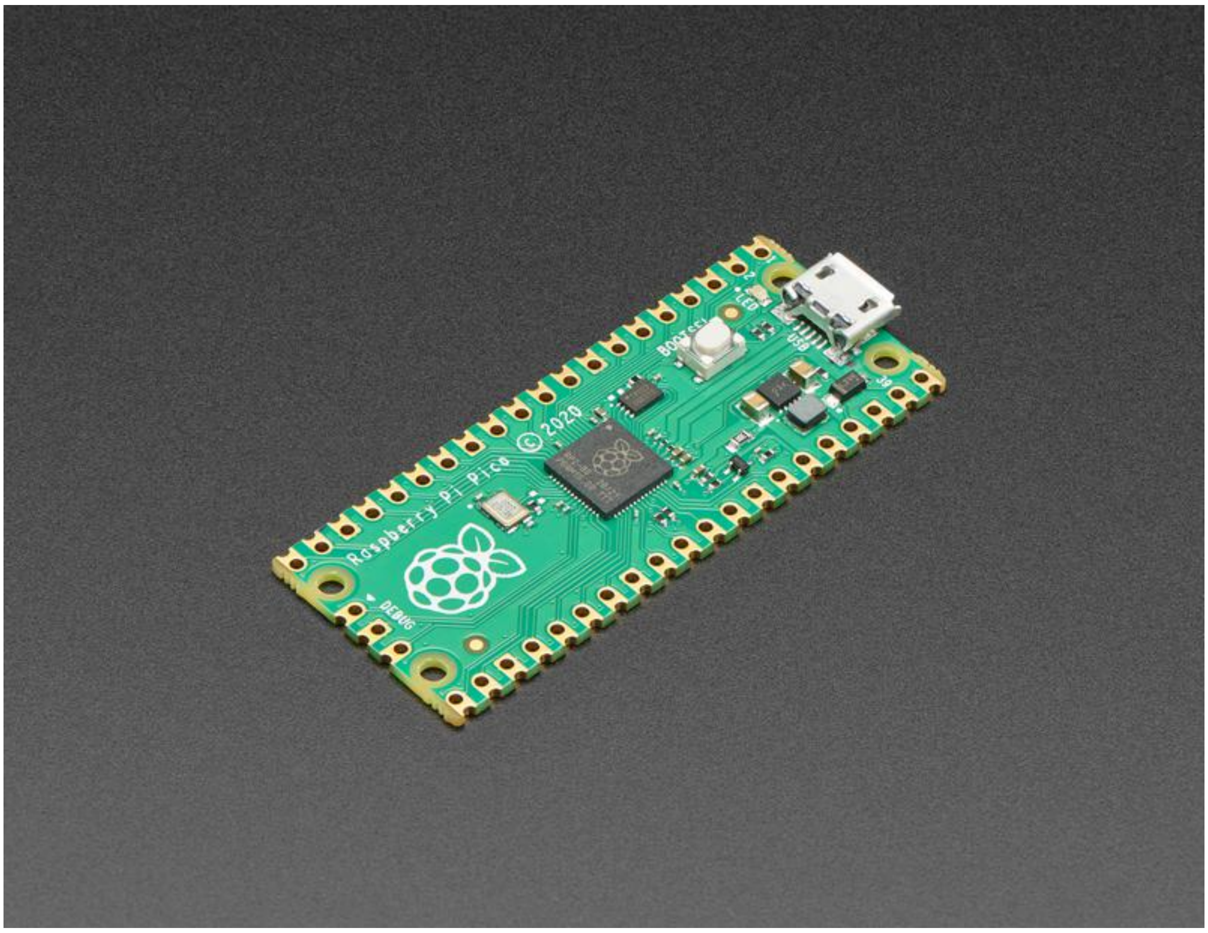
Getting Started with Raspberry Pi Pico and CircuitPython

Created by Kattni Rembor

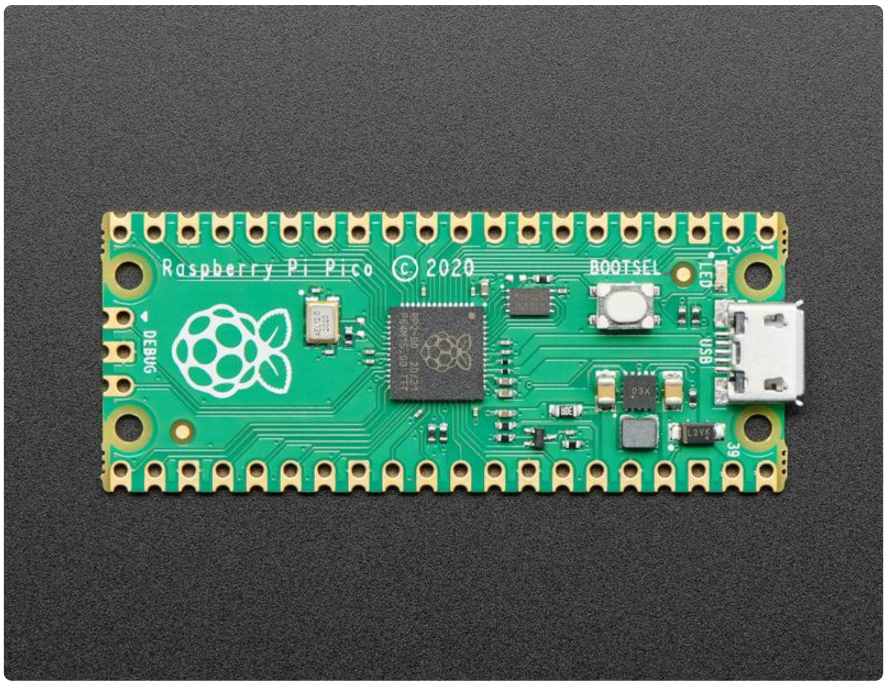


https://learn.adafruit.com/getting-started-with-raspberry-pi-pico-circuitpython

Last updated on 2023-11-08 04:19:25 PM EST

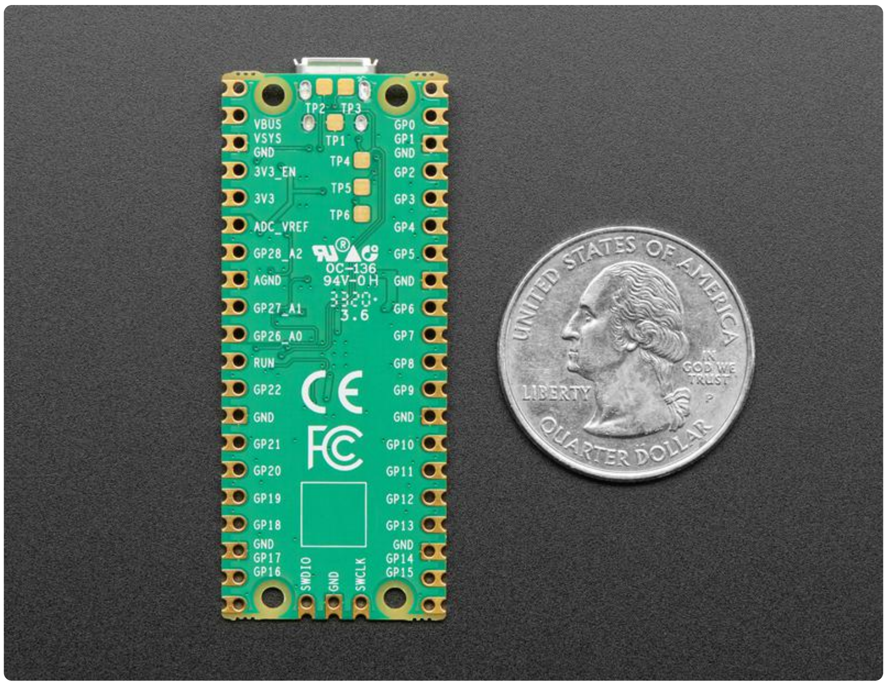
Table of Contents

Overview



The Raspberry Pi foundation changed single-board computing when they released the Raspberry Pi computer (), now they're ready to do the same for microcontrollers with the release of the brand new Raspberry Pi Pico. This low-cost microcontroller board features a powerful new chip, the RP2040, and all the fixin's to get started with embedded electronics projects at a stress-free price.

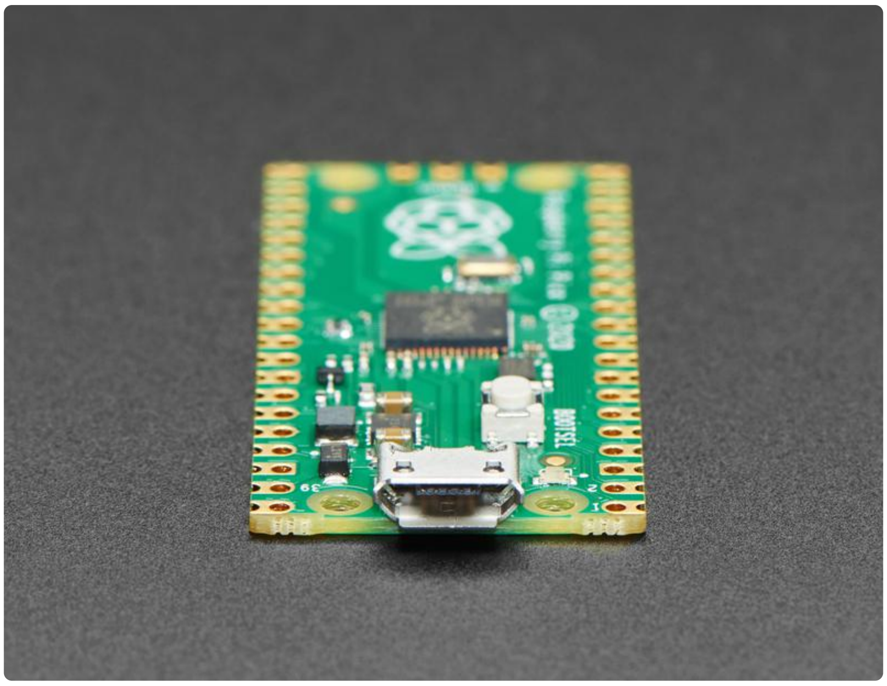
The Pico is 0.825" x 2" and can have headers soldered in for use in a breadboard or perfboard, or can be soldered directly onto a PCB with the castellated pads. There's 20 pads on each side, with groups of general purpose input-and-output (GPIO) pins interleaved with plenty of ground pins. All of the GPIO pins are 3.3V logic, and are not 5V-safe so stick to 3V! You get a total of 25 GPIO pins (technically there are 26 but IO #15 has a special purpose and should not be used by projects), 3 of those can be analog inputs (the chip has 4 ADC but one is not broken out). There are no true analog output (DAC) pins.



On the slim green board is minimal circuitry to get you going: A 5V to 3.3V power supply converter, single green LED on GP25, boot select button, RP2040 chip with dual-core Cortex M0+, 2 MegaBytes of QSPI flash storage, and crystal.

Inside the RP2040 is a 'permanent ROM' USB UF2 bootloader. What that means is when you want to program new firmware, you can hold down the BOOTSEL button while plugging it into USB (or pulling down the RUN/Reset pin to ground) and it will appear as a USB disk drive you can drag the firmware onto. Folks who have been using Adafruit products will find this very familiar - we use the technique on all our native-USB boards. Just note you don't double-click reset, instead hold down BOOTSEL during boot to enter the bootloader!

The RP2040 is a powerful chip, which has the clock speed of our M4 (SAMD51), and two cores that are equivalent to our M0 (SAMD21). Since it is an M0 chip, it does not have a floating point unit, or DSP hardware support - so if you're doing something with heavy floating point math, it will be done in software and thus not as fast as an M4. For many other computational tasks, you'll get close-to-M4 speeds!

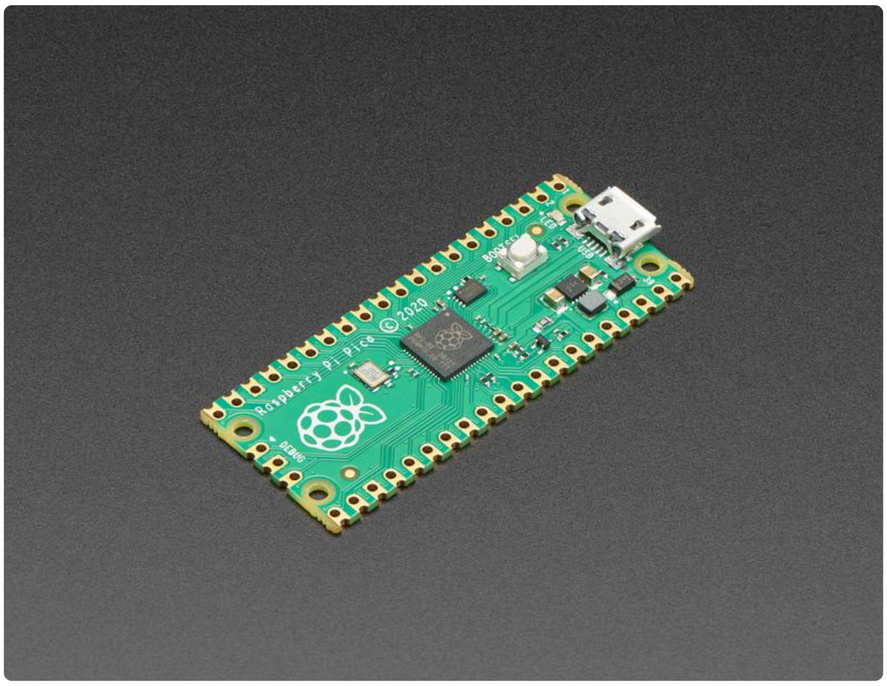


For peripherals, there are two I2C controllers, two SPI controllers, and two UARTs that are multiplexed across the GPIO - check the pinout for what pins can be set to which. There are 16 PWM channels, each pin has a channel it can be set to (ditto on the pinout).

You'll note there's no I2S peripheral, or SDIO, or camera, what's up with that? Well instead of having specific hardware support for serial-data-like peripherals like these, the RP2040 comes with the PIO state machine system which is a unique and powerful way to create custom hardware logic and data processing blocks that run on their own without taking up a CPU. For example, NeoPixels - often we bitbang the timing-specific protocol for these LEDs. For the RP2040, we instead use a PIO object that reads in the data buffer and clocks out the right bitstream with perfect accuracy. Same with I2S audio in or out, LED matrix displays, 8-bit or SPI based TFTs, even VGA ()! In MicroPython and CircuitPython you can create PIO control commands to script the peripheral and load it in at runtime. There are 2 PIO peripherals with 4 state machines each.

There is great C/C++ support (), Arduino ()support () (guide ()), an official MicroPython port (), and a CircuitPython port ()! We of course recommend CircuitPython because we think it's the easiest way to get started () and it has support with most our drivers, displays, sensors, and more, supported out of the box so you can follow along with our CircuitPython projects and tutorials.

While the RP2040 has lots of onboard RAM (264KB), it does not have built in FLASH memory. Instead that is provided by the external QSPI flash chip. On this board there is 2MB, which is shared between the program it's running and any file storage used by MicroPython or CircuitPython. When using C/C++ you get the whole flash memory, if using Python you will have about 1 MB remaining for code, files, images, fonts, etc.

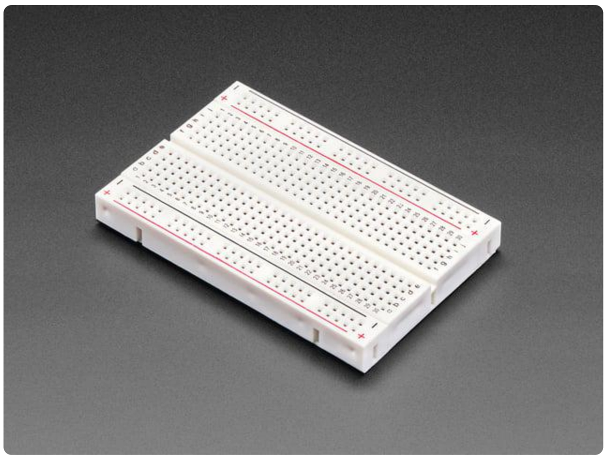


RP2040 Chip features:

- Dual ARM Cortex-M0+ @ 133MHz
- 264kB on-chip SRAM in six independent banks
- Support for up to 16MB of off-chip Flash memory via dedicated QSPI bus
- DMA controller
- Fully-connected AHB crossbar
- Interpolator and integer divider peripherals
- On-chip programmable LDO to generate core voltage
- 2 on-chip PLLs to generate USB and core clocks
- 30 GPIO pins, 3 of which can be used as analogue inputs
- Peripherals
- 2 UARTs
- 2 SPI controllers
- 2 I2C controllers
- 16 PWM channels
- USB 1.1 controller and PHY, with host and device support
- 8 PIO state machines

Other Required Hardware

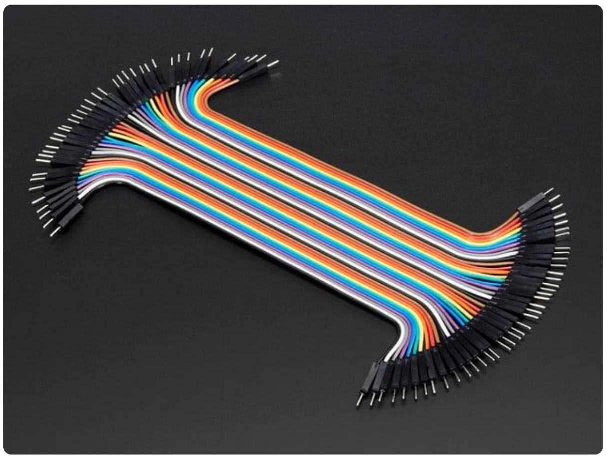
The following list of hardware, or some equivalent thereof, is required to complete this guide.



Half Sized Premium Breadboard - 400 Tie Points

This is a cute, half-size breadboard with 400 tie points, good for small projects. It's 3.25" x 2.2" / 8.3cm x 5.5cm with a standard double-strip in the...

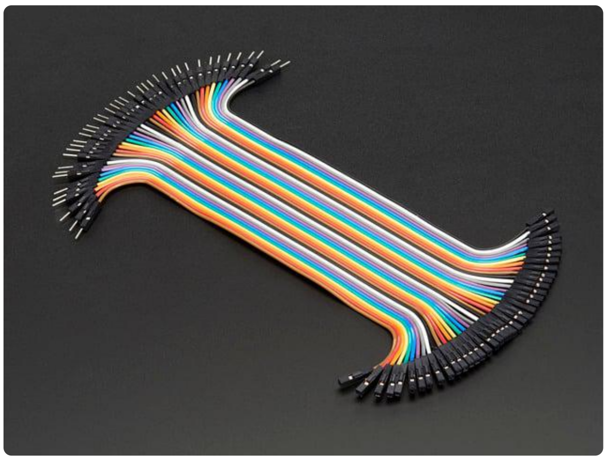
https://www.adafruit.com/product/64



Premium Male/Male Jumper Wires - 40 x 6" (150mm)

Handy for making wire harnesses or jumpering between headers on PCB's. These premium jumper wires are 6" (150mm) long and come in a 'strip' of 40 (4 pieces of each of...

https://www.adafruit.com/product/758



Premium Female/Male 'Extension' Jumper Wires - 40 x 6" (150mm)

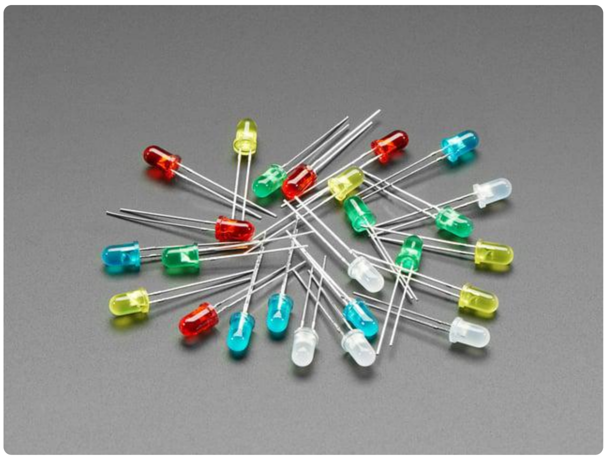
Handy for making wire harnesses or jumpering between headers on PCB's. These premium jumper wires are 6" (150mm) long and come in a 'strip' of 40 (4 pieces of each of...

https://www.adafruit.com/product/826

Tactile Switch Buttons (12mm square, 6mm tall) x 10 pack

Medium-sized clicky momentary switches are standard input "buttons" on electronic projects. These work best in a PCB but

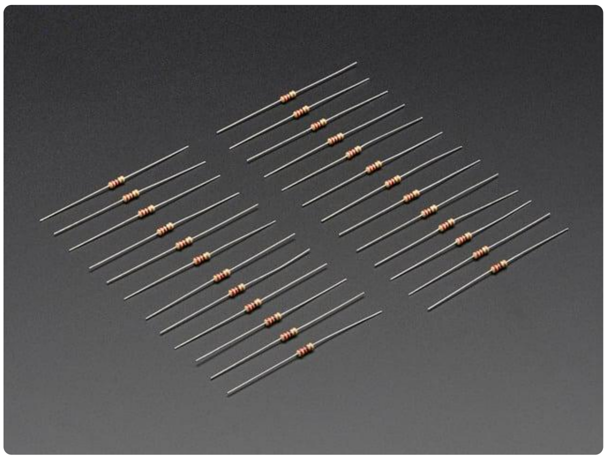
https://www.adafruit.com/product/1119



Diffused 5mm LED Pack - 5 LEDs each in 5 Colors - 25 Pack

Need some indicators? We are big fans of these diffused LEDs. They are fairly bright, so they can be seen in daytime, and from any angle. They go easily into a breadboard and will add...

https://www.adafruit.com/product/4203

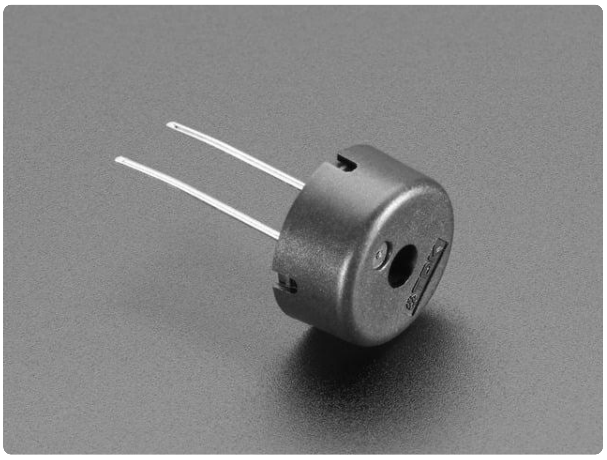


Through-Hole Resistors - 220 ohm 5% 1/4W - Pack of 25

ΩMG! You're not going to be able to resist these handy resistor packs! Well, axially, they do all of the resisting for you!This is a 25 Pack of...

https://www.adafruit.com/product/2780

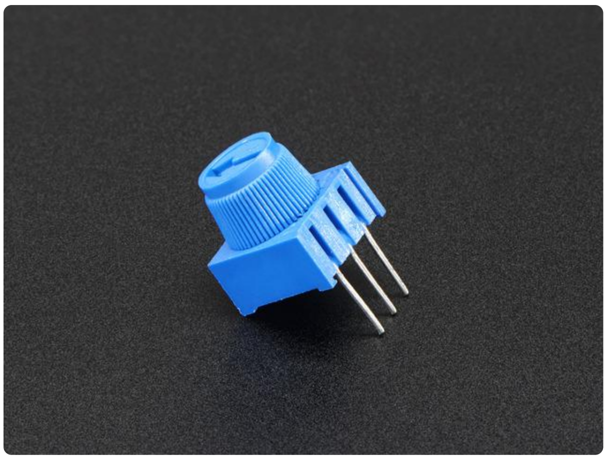
Any resistors with a value of 220Ω-1.0KΩ will work. The higher the value, the dimmer your LEDs will be!



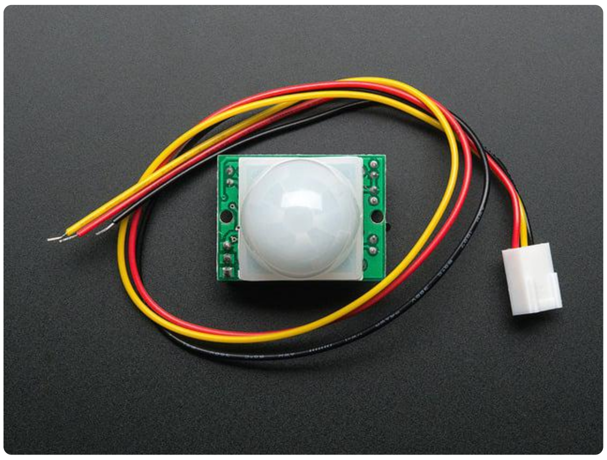
Piezo Buzzer

Piezo buzzers are used for making beeps, tones and alerts. This one is petite but loud! Drive it with 3-30V peak-to-peak square wave. To use, connect one pin to ground (either one) and...

https://www.adafruit.com/product/160



Breadboard trim potentiometer
These are our favorite trim pots, perfect for breadboarding and prototyping. They have a long grippy adjustment knob and with 0.1" spacing, they plug into breadboards or...
https://www.adafruit.com/product/356



PIR (motion) sensor
PIR sensors are used to detect motion from pets/humanoids from about 20 feet away (possibly works on zombies, not guaranteed). This one has an adjustable delay before firing (approx...
https://www.adafruit.com/product/189

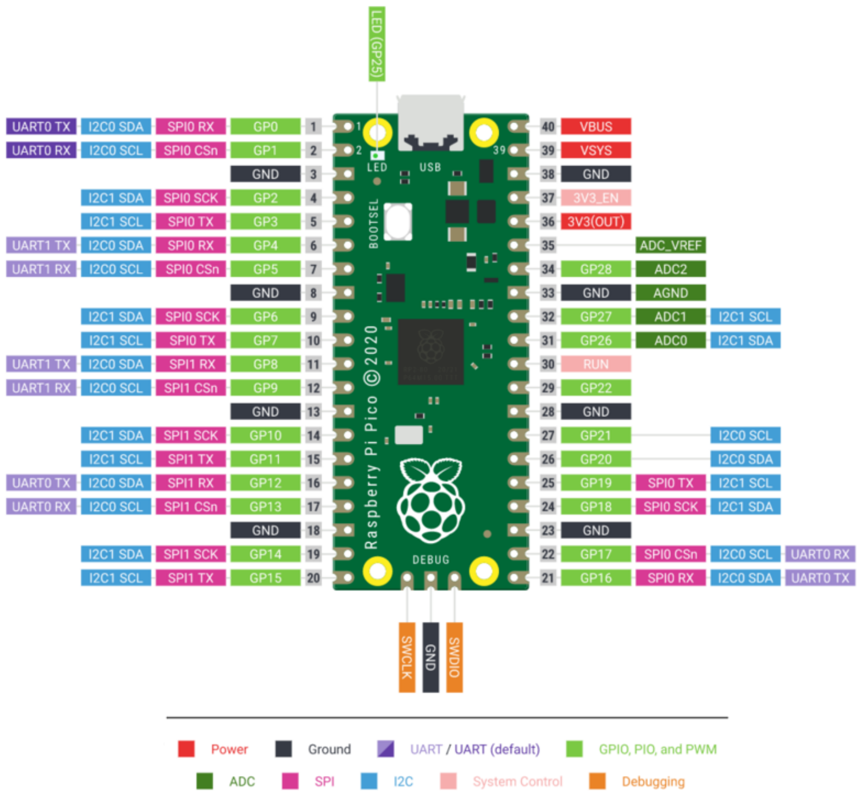
Adafruit NeoPixel LED Strip with 3-pin JST Connector - 1 meter
Plug in and glow, this Adafruit NeoPixel LED Strip with JST PH Connector has 30 total LEDs and is 1 meter long, in classy Adafruit...
https://www.adafruit.com/product/4801

Pinouts

This is a top view of the pinouts on the Raspberry Pi Pico. Click on the image for an enlarged, less blurry view. The pin labels are on the bottom of the board.

Another nice diagram is available at https://pico.pinout.xyz/ (). Click on "Advanced" to see extra information.

See the Downloads () page for a paper template you can put underneath the Pico to label the pins.



There are two I2C peripherals available, I2C0 and I2C1, two SPI peripherals, SPI0 and SPI1, and two UART peripherals, UART0 and UART1. You can assign any of these to the pins on which they are available. So for example, you can use GP0/GP1 for I2C0, and simultaneously use GP2/GP3 for I2C1, but you cannot use GP0/GP1 together with GP4/GP5 for I2C use, because they are both usable only with I2C0.

In CircuitPython, you don't need to specify exactly which peripheral is used. As long you choose valid pins, CircuitPython will choose a free peripheral.

No Basic Default `board` Devices

The Pico does not label specific pins as the defaults to use for I2C, SPI, or UART connections. So CircuitPython running on the Pico does not provide `board.I2C()`, `board.SPI()`, or `board.UART()`, since it's not immediately obvious what they would correspond to. For example:

CircuitPython for the Pico does include board.STEMMA_I2C() to work with the STEMMA I2C connector on the Adafruit Proto Cowbell. See the next section for details.

```
import board

i2c = board.I2C()    # Does not work on the Pico.
```

Instead, use the busio () module to create your bus and then specify the specific pins you want to use. To do so, use the pinout diagram above to find available pins, for example I2C0_SDA is on GP0 (as well as other locations). You then use the `board.GPx` pin name when creating the bus.

Here are some specific examples.

I2C Example

To setup an I2C bus (), you specify the SCL and SDA pins being used. You can look for "SCL" and "SDA" in the pin names in the pinout diagram above.

- I2Cx_SCL = SCL
- I2Cx_SDA = SDA

For example, here is how you would setup an I2C bus to use GP1 as SCL and GP0 as SDA:

```
import board
import busio

i2c = busio.I2C(scl=board.GP1, sda=board.GP0)
```

SPI Example

To setup a SPI bus (), you specify the SCK, MOSI (microcontroller out, sensor in), and MISO (microcontroller in, sensor out) pins. The Pico uses a different naming convention for these:

- SPIx_SCK = SCK
- SPIx_TX = MOSI
- SPIx_RX = MISO

So use that mapping to help find available pins.

Here's an example:

```
import board
import busio
```

```
spi = busio.SPI(clock=board.GP2, MOSI=board.GP3, MISO=board.GP4)
```

UART Example

To setup a UART bus (), you specify the TX and RX pins. Be sure to use the UARTx pins and not the similarly named SPIx ones.

Here's an example:

```
import board
import busio

uart = busio.UART(tx=board.GP4, rx=board.GP5)
```

The `board.STEMMA_I2C()` Object

CircuitPython running on the Pico includes the `board.STEMMA_I2C()` object. This represents a STEMMA QT connector connected to IO4 (SDA), and IO5 (SCL).

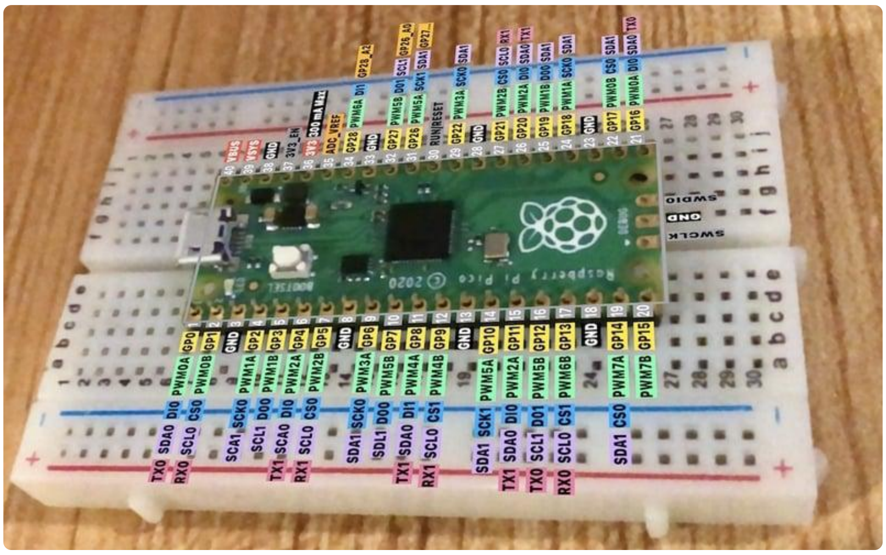
You can use it in your code when using the Adafruit PiCowbell Proto for Pico () with a STEMMA QT breakout connected to the included STEMMA QT connector.

Power Sensing Pins Available in CircuitPython

There are two pins available in CircuitPython for power sensing on the Pico.

- `VOLTAGE_MONITOR` (GP24) - This pin is connected to Vsys, which feed the voltage regulator, via a voltage divider. You can connect a battery to Vsys, and the voltage monitor pin can be used as a crude battery voltage monitor.
- `VBUS_SENSE` (GP29) - This pin is high if the board is powered by USB.

Using Adafruit AR with Raspberry Pi Pico



With this new Adafruit AR update, you’re able to scan your Raspberry Pi Pico to display an overlay of the boards pinouts and power pins. Here we'll show you how to get started.

To get started:

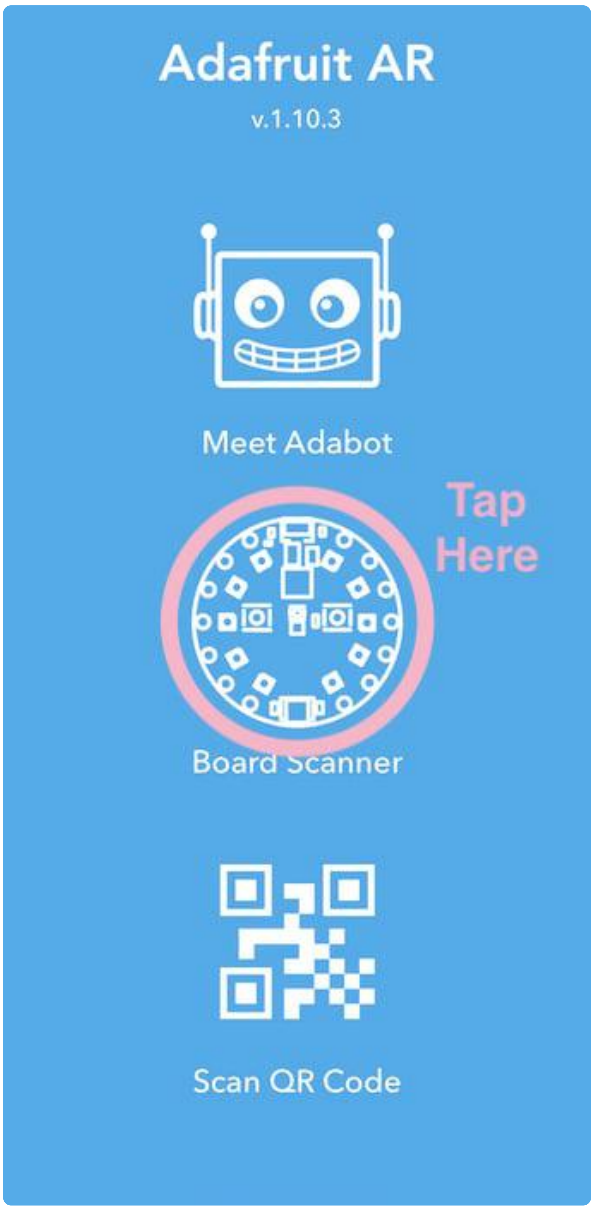
Before downloading this app, make sure your mobile device is running iOS 14 or greater.

Download Adafruit AR from the Apple App

[Adafruit AR on the App Store]()

Please note - the Adafruit AR app is currently only available for iOS.

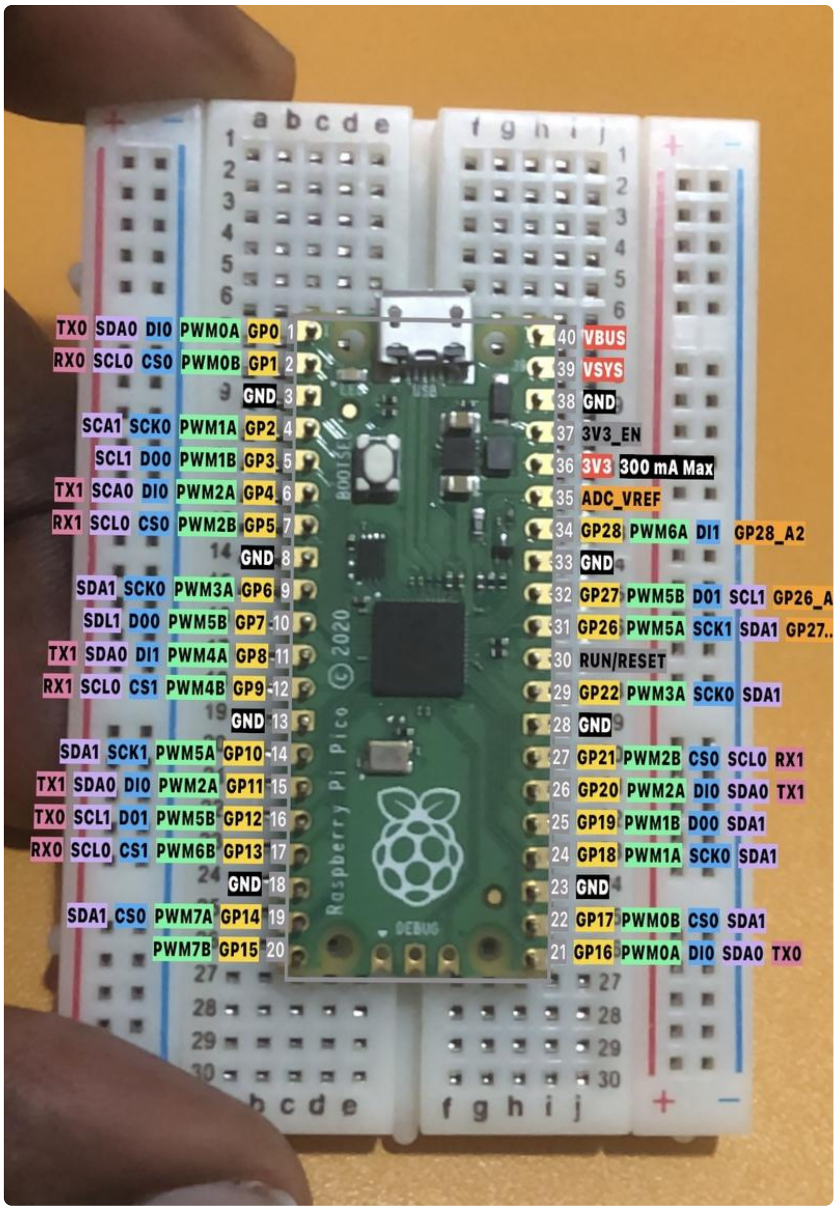
Once you’ve opened the app use the boards scanner mode to start.



With your Raspberry Pi Pico, scan the front of the board.

Due to the small size of the Raspberry Pi Pico, you'll need to scan the board a bit closely. Your device should be ideally 3 inches away from board to be recognized.

For the best practice to scan your board - make sure you are scanning in a well lit area and avoid light glare while scanning.



MicroPython or CircuitPython?

Now that you have a Pico you're probably wondering why most Adafruit tutorials are for CircuitPython why the official Python is called MicroPython?

Whats the difference, why use one or the other?

CircuitPython is a 'fork' based on MicroPython

CircuitPython code is gonna look a lot like MicroPython because they're based on the same Python implementation. There are some minor differences because CircuitPython is focused on beginners, education, and getting folks started super fast. If you already know MicroPython, 95% of what you know will work just the same!

CircuitPython is also available for the Pico and generally RP2040 boards. You load it just like loading MicroPython.

While CircuitPython is based on MicroPython, there are some key differences why you may want to use CircuitPython instead of MicroPython.

So what's different?

There's a few differences and they're all documented here (), however for Pico users who have tried or are following MicroPython guides the most important are...

CircuitPython was designed to have a USB disk drive that appears when you plug in the board. That disk drive is small (on the order of MB!) and holds your code and files. You can treat it just like a disk drive - drag and drop files, delete and copy them. You do not need to use Thonny to 'upload' a file - simply drag any file you want to the USB drive.

CircuitPython will restart your code when you save files to the disk drive. That means when you write Python code, whenever you save it will auto-reload the code for you, for instant gratification. This is a little unusual for programmers who are used to 'edit-save-compile-upload-reset-run' - we go straight to 'edit-save-run'.

CircuitPython has a consistent API across all boards. That means that whether you're using a Pico, or an nRF52840 or an ESP32-S2 or SAMD51 for your project, the code for your hardware is identical. (Other than pin names which may vary depending on how many there are on the board itself and what they're called).

CircuitPython has a lot of examples and support!
There are 260+ libraries for the standard CircuitPython API. Most of these will already work. Listed here ()

Tons of guides and tutorials at https://learn.adafruit.com/category/circuitpython ()

Most CircuitPython libraries also work on Raspberry Pis via the Blinka library (). That means you can write code that works on both!

Why Use MicroPython?

You may want to use MicroPython for:
1) Advanced APIs such as interrupts and threading.

2) Complete PIO API (CircuitPython's support is incomplete)
3) Using existing MicroPython code

It's great to know both!

To get started quick:

Download CircuitPython for the Pico from circuitpython.org: https://circuitpython.org/board/raspberry_pi_pico/ ()

For now, click "Absolute Newest", then click your language code such as "en_US", and finally download the UF2 file at the top. That will be the latest and greatest version of CircuitPython. As support matures, the download page will stable releases. You can also choose a beta release but many features and fixes are being added on a daily basis, so "Absolute Newest" will be the best for a while.

After dragging the CircuitPython UF2 to RPI-RP2 bootloader the chip will reset and show a CIRCUITPY drive.

See the Welcome to CircuitPython () and CircuitPython Essentials () guides for CircuitPython basics. API Docs are here () though they won't include RP2040 specific modules until support is merged in. A Pico specific guide that will grow in time is here ().

Join the Adafruit Discord () for #help-with-circuitpython and feel free to mention @tannewt for RP2040 specific questions.

What is CircuitPython?

CircuitPython is a programming language designed to simplify experimenting and learning to program on low-cost microcontroller boards. It makes getting started easier than ever with no upfront desktop downloads needed. Once you get your board set up, open any text editor, and get started editing code. It's that simple.

CircuitPython is based on Python

Python is the fastest growing programming language. It's taught in schools and universities. It's a high-level programming language which means it's designed to be easier to read, write and maintain. It supports modules and packages which means it's easy to reuse your code for other projects. It has a built in interpreter which means there are no extra steps, like compiling, to get your code to work. And of course, Python is Open Source Software which means it's free for anyone to use, modify or improve upon.

CircuitPython adds hardware support to all of these amazing features. If you already have Python knowledge, you can easily apply that to using CircuitPython. If you have no previous experience, it's really simple to get started!

Why would I use CircuitPython?

CircuitPython is designed to run on microcontroller boards. A microcontroller board is a board with a microcontroller chip that's essentially an itty-bitty all-in-one computer. The board you're holding is a microcontroller board! CircuitPython is easy to use because all you need is that little board, a USB cable, and a computer with a USB connection. But that's only the beginning.

Other reasons to use CircuitPython include:

- You want to get up and running quickly. Create a file, edit your code, save the file, and it runs immediately. There is no compiling, no downloading and no uploading needed.
- You're new to programming. CircuitPython is designed with education in mind. It's easy to start learning how to program and you get immediate feedback from the board.

- Easily update your code. Since your code lives on the disk drive, you can edit it whenever you like, you can also keep multiple files around for easy experimentation.
- The serial console and REPL. These allow for live feedback from your code and interactive programming.
- File storage. The internal storage for CircuitPython makes it great for data-logging, playing audio clips, and otherwise interacting with files.
- Strong hardware support. CircuitPython has builtin support for microcontroller hardware features like digital I/O pins, hardware buses (UART, I2C, SPI), audio I/O, and other capabilities. There are also many libraries and drivers for sensors, breakout boards and other external components.
- It's Python! Python is the fastest-growing programming language. It's taught in schools and universities. CircuitPython is almost-completely compatible with Python. It simply adds hardware support.

This is just the beginning. CircuitPython continues to evolve, and is constantly being updated. Adafruit welcomes and encourages feedback from the community, and incorporate it into the development of CircuitPython. That's the core of the open source concept. This makes CircuitPython better for you and everyone who uses it!

Installing CircuitPython

CircuitPython () is a derivative of MicroPython () 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.

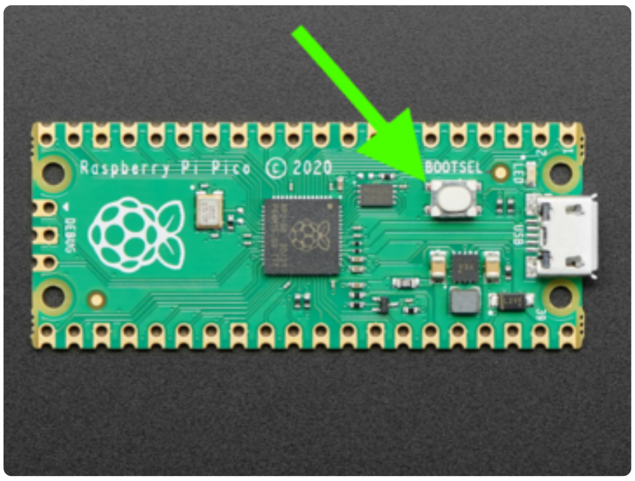
CircuitPython Quickstart

Follow this step-by-step to quickly get CircuitPython working on your board.

Download the latest version of CircuitPython for the Raspberry Pi Pico from circuitpython.org

Click the link above and download the latest UF2 file.

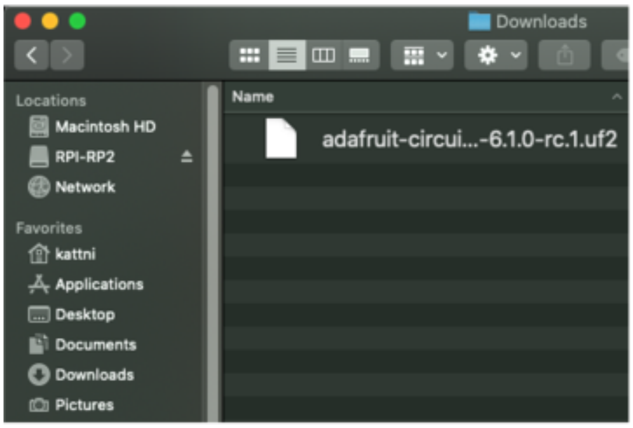
Download and save it to your desktop (or wherever is handy).



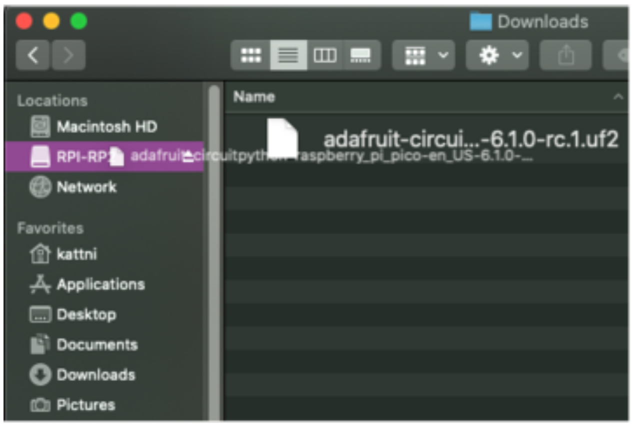
Start with your Pico unplugged from USB. Hold down the BOOTSEL button, and while continuing to hold it (don't let go!), plug the Pico into USB. Continue to hold the BOOTSEL button until the RPI-RP2 drive appears!

If the drive does not appear, unplug your Pico and go through the above process again.

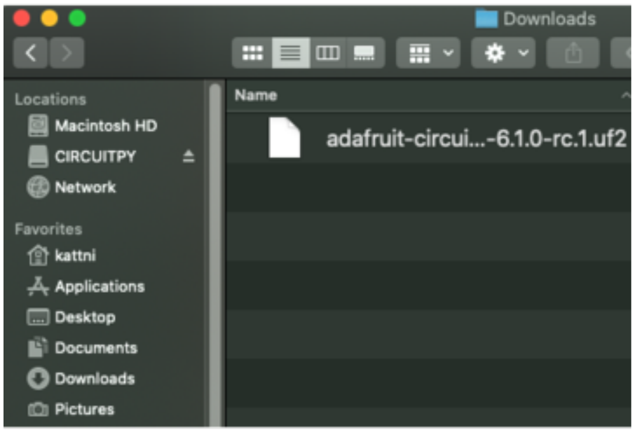
A lot of people end up using charge-only USB cables and it is very frustrating! So make sure you have a USB cable you know is good for data sync.



You will see a new disk drive appear called RPI-RP2.



Drag the adafruit_circuitpython_etc.uf2 file to RPI-RP2.



The RPI-RP2 drive will disappear and a new disk drive called CIRCUITPY will appear.

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

Flash Resetting UF2

If your Pico ever gets into a really weird state and doesn't even show up as a disk drive when installing CircuitPython, try installing this 'nuke' UF2 which will do a 'deep clean' on your Flash Memory. You will lose all the files on the board, but at least you'll be able to revive it! After nuking, re-install CircuitPython

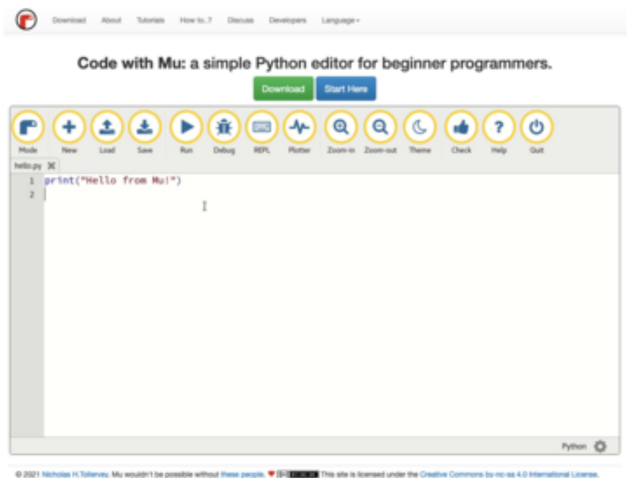
flash_nuke.uf2

Installing the Mu Editor

Mu is a simple code editor that works with the Adafruit CircuitPython boards. It's written in Python and works on Windows, MacOS, Linux and Raspberry Pi. The serial console is built right in so you get immediate feedback from your board's serial output!

Mu is our recommended editor - please use it (unless you are an experienced coder with a favorite editor already!).

Download and Install Mu



Code with Mu: a simple Python editor for beginner programmers.

```
print("Hello from Mu!")
```

© 2021 Nicholas H.Tollervey. Mu wouldn't be possible without these people. ♥ This site is licensed under the Creative Commons by-nc-sa 4.0 International License.

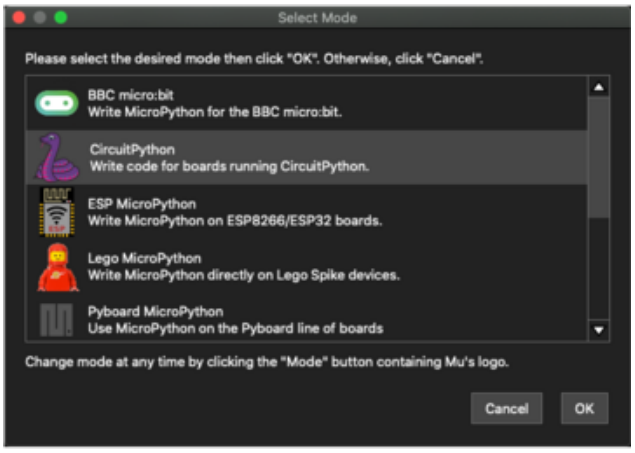
Download Mu from https://codewith.mu ().

Click the Download link for downloads and installation instructions.

Click Start Here to find a wealth of other information, including extensive tutorials and and how-to's.

Windows users: due to the nature of MSI installers, please remove old versions of Mu before installing the latest version.

Starting Up Mu


Select Mode

Please select the desired mode then click "OK". Otherwise, click "Cancel".

BBC micro:bit
Write MicroPython for the BBC micro:bit.

CircuitPython
Write code for boards running CircuitPython.

ESP MicroPython
Write MicroPython on ESP8266/ESP32 boards.

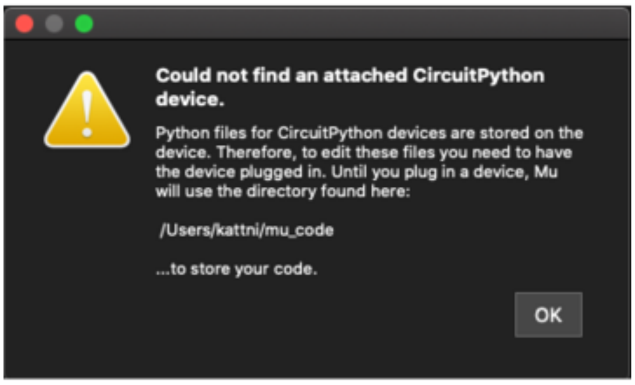
Lego MicroPython
Write MicroPython directly on Lego Spike devices.

Pyboard MicroPython
Use MicroPython on the Pyboard line of boards

Change mode at any time by clicking the "Mode" button containing Mu's logo.

The first time you start Mu, you will be prompted to select your 'mode' - you can always change your mind later. For now please select CircuitPython!

The current mode is displayed in the lower right corner of the window, next to the "gear" icon. If the mode says "Microbit" or something else, click the Mode button in the upper left, and then choose "CircuitPython" in the dialog box that appears.


Could not find an attached CircuitPython device.

Python files for CircuitPython devices are stored on the device. Therefore, to edit these files you need to have the device plugged in. Until you plug in a device, Mu will use the directory found here:

/Users/kattni/mu_code

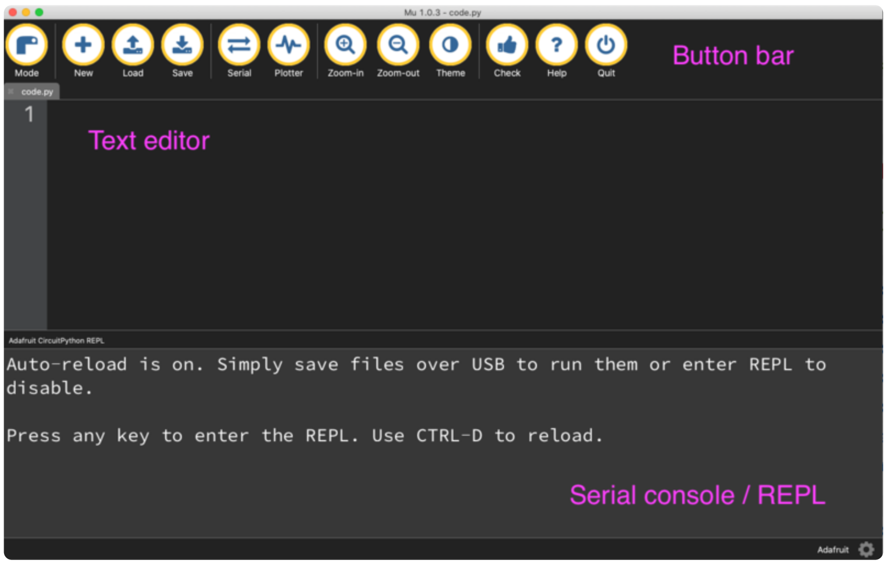
...to store your code.

Mu attempts to auto-detect your board on startup, so if you do not have a CircuitPython board plugged in with a CIRCUITPY drive available, Mu will inform you where it will store any code you save until you plug in a board.

To avoid this warning, plug in a board and ensure that the CIRCUITPY drive is mounted before starting Mu.

Using Mu

You can now explore Mu! The three main sections of the window are labeled below; the button bar, the text editor, and the serial console / REPL.



```
Auto-reload is on. Simply save files over USB to run them or enter REPL to
disable.

Press any key to enter the REPL. Use CTRL-D to reload.
```

Now you're ready to code! Let's keep going...

CircuitPython Programming Basics

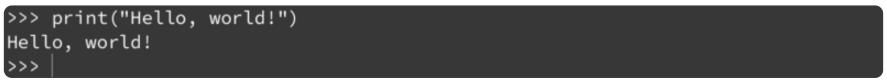
To get you started with how to program your Pico in CircuitPython, especially for those who may have started out with the official MicroPython setup, we've 'ported' the Getting Started with MicroPython on Pico book () examples to CircuitPython. The book is awesome, please download/purchase it to support Raspberry Pi Press ()!

Now that you've installed CircuitPython, it's time to begin your first program. If you haven't already, plug in your Raspberry Pi Pico to your computer via USB. Then, open your favorite Python editor.

Click the Serial button in Mu to open the serial console. Click anywhere in the serial output window at the bottom of Mu, and press CTRL+C on your keyboard. This will bring you to the REPL. At the REPL prompt (`>>>`), type the following code followed by the ENTER key:

```
print("Hello, world!")
```

Immediately upon pressing enter, the code is executed. Python responds by printing the message.



```
>>> print("Hello, world!")
Hello, world!
>>>
```

The REPL, or the Read-Evaluate-Print-Loop, allows you to run individual lines of code, one at a time. You can run multiple lines of code in sequence to execute a longer program. It's great for testing a program line by line to determine where an issue might be. It's interactive, so it's excellent for testing new ideas.

It is, however, important to remember that the REPL is ephemeral. Any code you write there is not saved anywhere. If you'd like to save your code, you can easily do so with CircuitPython and Mu.

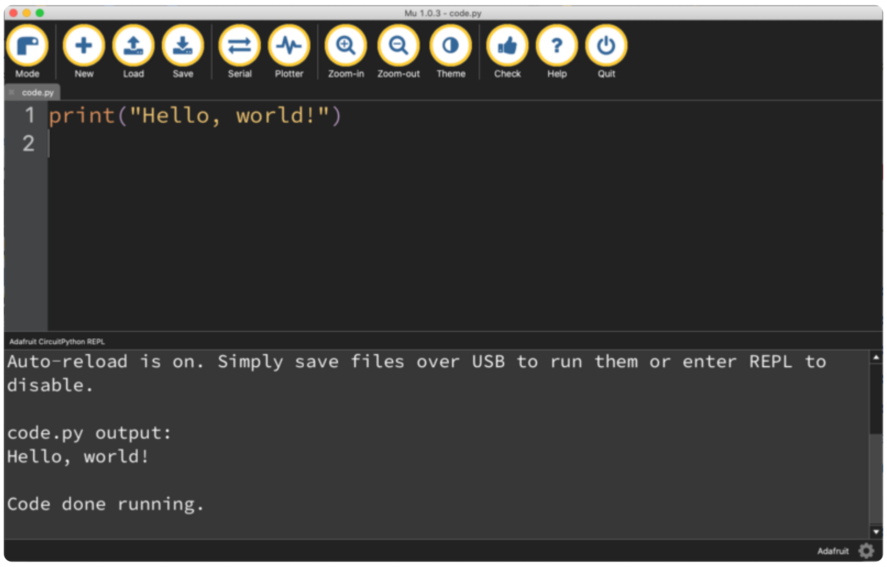
Once installed, CircuitPython presents your Pico board as a USB drive called CIRCUIT PY. Your code and any necessary libraries live on this drive. With a fresh CircuitPython install, you'll find a code.py file containing `print("Hello World!")` and an empty li b folder. If your CIRCUITPY drive does not contain a code.py file, you can easily create one and save it to the drive. CircuitPython looks for code.py and executes the code within the file automatically when the board starts up or resets. Following a change to the contents of CIRCUITPY, such as making a change to the code.py file, the board will reset, and the code will be run. You do not need to manually run the code. Note that all changes to the contents of CIRCUITPY, such as saving a new file, renaming a current file, or deleting an existing file will trigger a reset of the board.

In Mu, if the file currently open is not code.py, click Load in the Mu button bar, navigate to your CIRCUITPY drive, and open code.py. If there is no code.py file, create a new file by clicking New in the Mu button bar, then click Save, navigate to your CIRCUITPY drive, and save the file as code.py.

Note that the code in code.py is not running while you are actively in the REPL. To exit the REPL, simply type CTRL+D at the REPL prompt (`>>>`).

You must exit the REPL for the code found in code.py to be run.

In the code.py file, click in the Mu text editor, and add the same simple line of code as above (if it's not already there). Save the file. The code will run automatically, and the message will be printed in the serial console!



```
print("Hello, world!")
```

```
Auto-reload is on. Simply save files over USB to run them or enter REPL to
disable.

code.py output:
Hello, world!

Code done running.
```

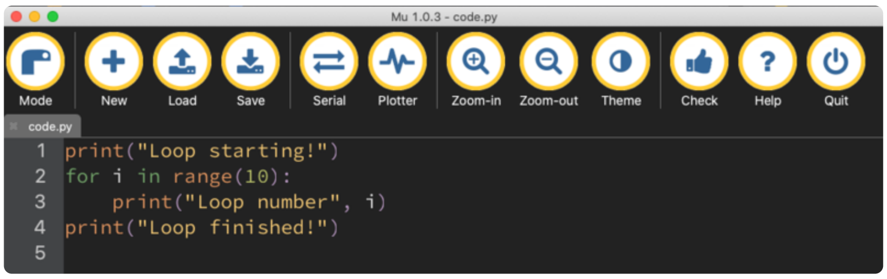
Indentation and Code Loops

CircuitPython runs the same as a standard Python program, typically running from top to bottom, executing each line. However, you can control the flow of a program using indentation. Delete the current contents of your code.py file and replace them with the following code:

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

"""Example of definite loop."""
print("Loop starting!")
for i in range(10):
    print("Loop number", i)
print("Loop finished!")
```

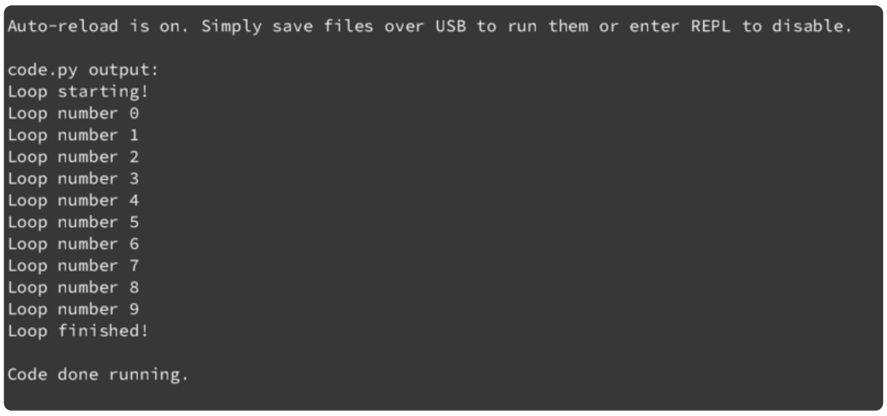
Loop is referring to a section of code that runs repeatedly - in this instance, the `for i in range(10):`. This is called a definite loop, a loop that runs a set number of times, in this case, 10.

Your code.py should look like this:



```
print("Loop starting!")
for i in range(10):
    print("Loop number", i)
print("Loop finished!")
```

Save the file, and check out the serial output.



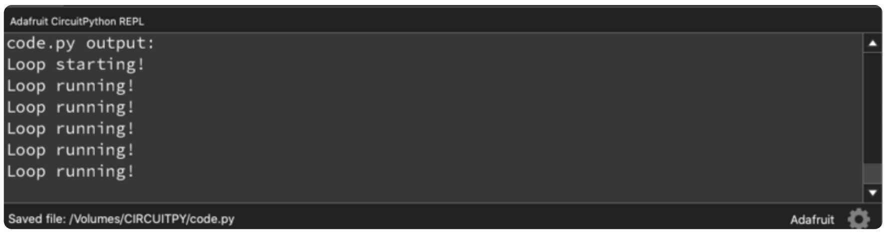
```
Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.

code.py output:
Loop starting!
Loop number 0
Loop number 1
Loop number 2
Loop number 3
Loop number 4
Loop number 5
Loop number 6
Loop number 7
Loop number 8
Loop number 9
Loop finished!

Code done running.
```

Indentation is crucial to flow control in code, but is also a very common cause of syntax errors, which causes the code to fail to run. If your code fails with a syntax error, be sure to check your indentation.

There are also indefinite loops in CircuitPython, that is, a loop that continues indefinitely. Update your code.py to the following. Be sure to delete the existing code! Your code.py file should only include the following code.

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

"""Example of infinite loop. Final print statement is never reached."""
print("Loop starting!")
while True:
    print("Loop running!")
print("Loop finished!")
```

Keep an eye on the serial console and click save. You should see the `Loop Starting!` message posted once initially, and then the `Loop running!` message repeated indefinitely.


Adafruit CircuitPython REPL
Saved file: /Volumes/CIRCUITPY/code.py
Adafruit

```
code.py output:
Loop starting!
Loop running!
Loop running!
Loop running!
Loop running!
Loop running!
```

The `Loop Finished!` message will never show up because it is not "inside" the loop. When there is a `while True:` in Python code, everything indented under it will run repeatedly - when the end of the loop is reached, it will begin again at the beginning of the loop.

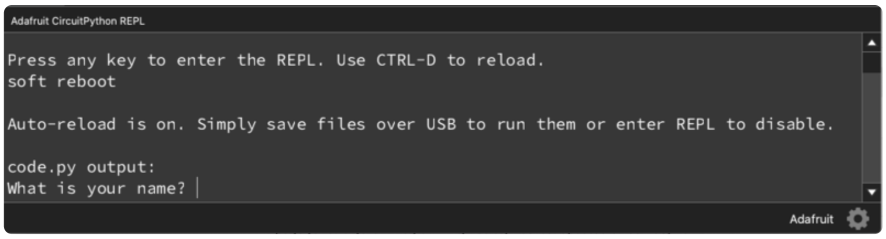
Conditionals and Variables

In CircuitPython, like Python, you can create variables. You can think of variables as a name attached to a specific object. To create a variable, you simply assign it and start using it. You use `=` to assign a variable name to the desired object.

Update your code.py to the following.

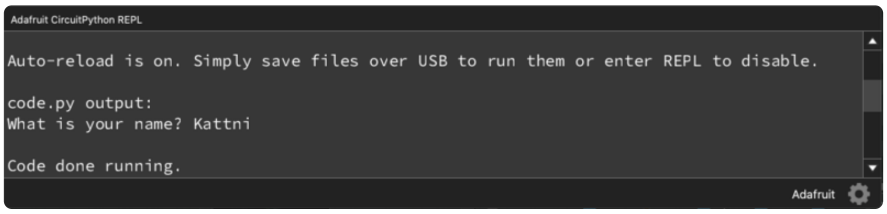
```
# SPDX-FileCopyrightText: 2021 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""Example of assigning a variable."""
user_name = input("What is your name? ")
```

Save the file.



```
Press any key to enter the REPL. Use CTRL-D to reload.
soft reboot

Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.

code.py output:
What is your name?
```

Click into the serial output. Type your answer to the question and press ENTER on your keyboard.



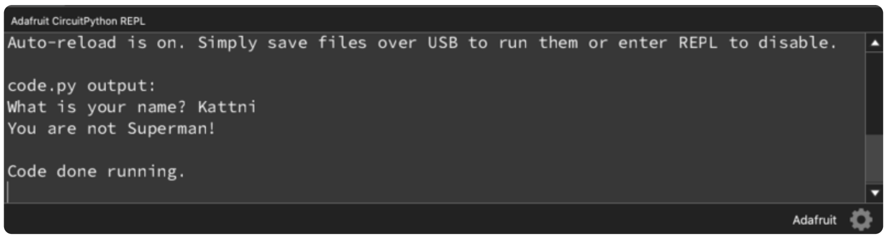
```
Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.

code.py output:
What is your name? Kattni

Code done running.
```

You've saved your name to the `user_name` variable! Now you can do something with it.

Update code.py to the following. Note that when Mu sees that your code needs to be indented, it will do it automatically. Therefore, if your next line of code does not need to be indented, you'll need to backspace to remove the indentation.

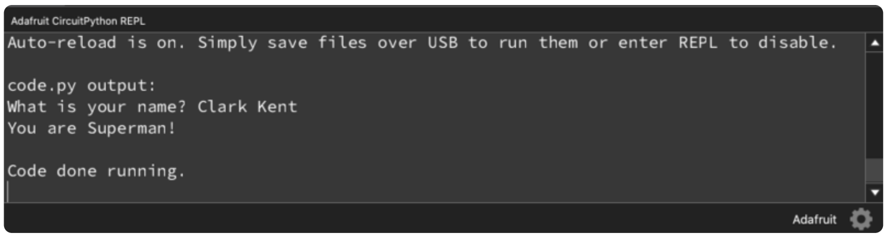
```
# SPDX-FileCopyrightText: 2021 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""Example of assigning a variable and comparing it to a value."""
user_name = input ("What is your name? ")

if user_name == "Clark Kent":
    print("You are Superman!")
else:
    print("You are not Superman!")
```

Save your code.py file. Now click in the serial console, type in your name, and press ENTER on your keyboard. Unless your name is "Clark Kent", you'll see the `You are not Superman!` message.



```
Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.

code.py output:
What is your name? Kattni
You are not Superman!

Code done running.
```

While still in the serial console, type CTRL+D to reload, and run the code again. This time, type in `Clark Kent`, and press ENTER on your keyboard. Make sure you have the capitalisation exactly as shown!



```
Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.

code.py output:
What is your name? Clark Kent
You are Superman!

Code done running.
```

You are Superman!

The `==` symbol tells CircuitPython to directly compare the text entered at the prompt, also known as a string, with "Clark Kent" to see if they are the same. If they are, then it prints the first message, `You are Superman!`, found under the `if` statement. If they are not, it prints the second message, `You are not Superman!`, found under the `else` statement.

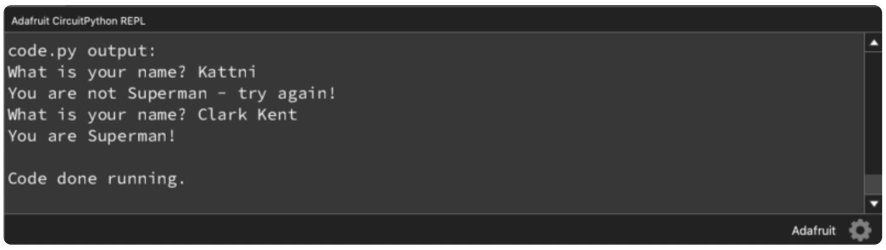
There are other symbols, such as `>` and `>=`, to use if you're working with numbers instead of strings. To check if a string or value is not the same as another string or value, you would use `!=`, which is essentially the opposite of `==`. These symbols are collectively known as comparison operators.

Note that `=` sets a variable equal to the value following it, and `==` checks to see if the variable is equal to the value following it. Don't mix them up!

Comparison operators can be used in loops as well. Update your code.py to the following and save.

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

"""Example of assigning a variable, and comparing it to a value in a loop."""
user_name = input ("What is your name? ")

while user_name != "Clark Kent":
    print("You are not Superman - try again!")
    user_name = input ("What is your name? ")
print("You are Superman!")
```

This time, instead of the code ending following the comparison of the entered name to Clark Kent, it will keep asking for your name until it turns out that you're Superman, at which point it will stop running.



```
code.py output:
What is your name? Kattni
You are not Superman - try again!
What is your name? Clark Kent
You are Superman!

Code done running.
```

That's only the beginning of what you can do with variables and conditionals in CircuitPython!

CircuitPython Pins and Modules

CircuitPython is designed to run on microcontrollers and allows you to interface with all kinds of sensors, inputs and other hardware peripherals. There are tons of guides showing how to wire up a circuit, and use CircuitPython to, for example, read data from a sensor, or detect a button press. Most CircuitPython code includes hardware setup which requires various modules, such as `board` or `digitalio`. You import these modules and then use them in your code. How does CircuitPython know to look for hardware in the specific place you connected it, and where do these modules come from?

This page explains both. You'll learn how CircuitPython finds the pins on your microcontroller board, including how to find the available pins for your board and what each pin is named. You'll also learn about the modules built into CircuitPython, including how to find all the modules available for your board.

CircuitPython Pins

When using hardware peripherals with a CircuitPython compatible microcontroller, you'll almost certainly be utilising pins. This section will cover how to access your board's pins using CircuitPython, how to discover what pins and board-specific objects are available in CircuitPython for your board, how to use the board-specific objects, and how to determine all available pin names for a given pin on your board.

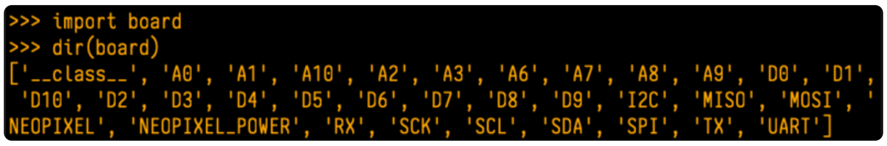
`import board`

When you're using any kind of hardware peripherals wired up to your microcontroller board, the import list in your code will include `import board`. The `board` module is built into CircuitPython, and is used to provide access to a series of board-specific objects, including pins. Take a look at your microcontroller board. You'll notice that next to the pins are pin labels. You can always access a pin by its pin label. However, there are almost always multiple names for a given pin.

To see all the available board-specific objects and pins for your board, enter the REPL (`>>>`) and run the following commands:

```
import board
dir(board)
```

Here is the output for the QT Py SAMD21. You may have a different board, and this list will vary, based on the board.



```
>>> import board
>>> dir(board)
['__class__', 'A0', 'A1', 'A10', 'A2', 'A3', 'A6', 'A7', 'A8', 'A9', 'D0', 'D1',
 'D10', 'D2', 'D3', 'D4', 'D5', 'D6', 'D7', 'D8', 'D9', 'I2C', 'MISO', 'MOSI', '
NEOPIXEL', 'NEOPIXEL_POWER', 'RX', 'SCK', 'SCL', 'SDA', 'SPI', 'TX', 'UART']
```

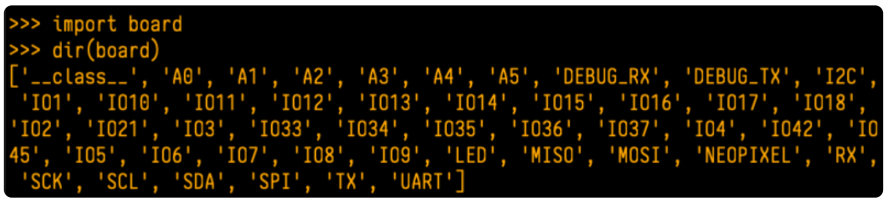
The following pins have labels on the physical QT Py SAMD21 board: A0, A1, A2, A3, SDA, SCL, TX, RX, SCK, MISO, and MOSI. You see that there are many more entries available in `board` than the labels on the QT Py.

You can use the pin names on the physical board, regardless of whether they seem to be specific to a certain protocol.

For example, you do not have to use the SDA pin for I2C - you can use it for a button or LED.

On the flip side, there may be multiple names for one pin. For example, on the QT Py SAMD21, pin A0 is labeled on the physical board silkscreen, but it is available in CircuitPython as both `A0` and `D0`. For more information on finding all the names for a given pin, see the What Are All the Available Pin Names? () section below.

The results of `dir(board)` for CircuitPython compatible boards will look similar to the results for the QT Py SAMD21 in terms of the pin names, e.g. A0, D0, etc. However, some boards, for example, the Metro ESP32-S2, have different styled pin names. Here is the output for the Metro ESP32-S2.



```
>>> import board
>>> dir(board)
['__class__', 'A0', 'A1', 'A2', 'A3', 'A4', 'A5', 'DEBUG_RX', 'DEBUG_TX', 'I2C',
 'IO1', 'IO10', 'IO11', 'IO12', 'IO13', 'IO14', 'IO15', 'IO16', 'IO17', 'IO18',
'IO2', 'IO21', 'IO3', 'IO33', 'IO34', 'IO35', 'IO36', 'IO37', 'IO4', 'IO42', 'IO
45', 'IO5', 'IO6', 'IO7', 'IO8', 'IO9', 'LED', 'MISO', 'MOSI', 'NEOPIXEL', 'RX',
 'SCK', 'SCL', 'SDA', 'SPI', 'TX', 'UART']
```

Note that most of the pins are named in an IO# style, such as IO1 and IO2. Those pins on the physical board are labeled only with a number, so an easy way to know how to access them in CircuitPython, is to run those commands in the REPL and find the pin naming scheme.

If your code is failing to run because it can't find a pin name you provided, verify that you have the proper pin name by running these commands in the REPL.

I2C, SPI, and UART

You'll also see there are often (but not always!) three special board-specific objects included: `I2C`, `SPI`, and `UART` - each one is for the default pin-set used for each of the three common protocol busses they are named for. These are called singletons.

What's a singleton? When you create an object in CircuitPython, you are instantiating ('creating') it. Instantiating an object means you are creating an instance of the object with the unique values that are provided, or "passed", to it.

For example, When you instantiate an I2C object using the `busio` module, it expects two pins: clock and data, typically SCL and SDA. It often looks like this:

```
i2c = busio.I2C(board.SCL, board.SDA)
```

Then, you pass the I2C object to a driver for the hardware you're using. For example, if you were using the TSL2591 light sensor and its CircuitPython library, the next line of code would be:

```
tsl2591 = adafruit_tsl2591.TSL2591(i2c)
```

However, CircuitPython makes this simpler by including the `I2C` singleton in the `board` module. Instead of the two lines of code above, you simply provide the singleton as the I2C object. So if you were using the TSL2591 and its CircuitPython library, the two above lines of code would be replaced with:

```
tsl2591 = adafruit_tsl2591.TSL2591(board.I2C())
```

The board.I2C(), board.SPI(), and board.UART() singletons do not exist on all boards. They exist if there are board markings for the default pins for those devices.

This eliminates the need for the `busio` module, and simplifies the code. Behind the scenes, the `board.I2C()` object is instantiated when you call it, but not before, and on subsequent calls, it returns the same object. Basically, it does not create an object until you need it, and provides the same object every time you need it. You can call `board.I2C()` as many times as you like, and it will always return the same object.

The UART/SPI/I2C singletons will use the 'default' bus pins for each board - often labeled as RX/TX (UART), MOSI/MISO/SCK (SPI), or SDA/SCL (I2C). Check your board documentation/pinout for the default busses.

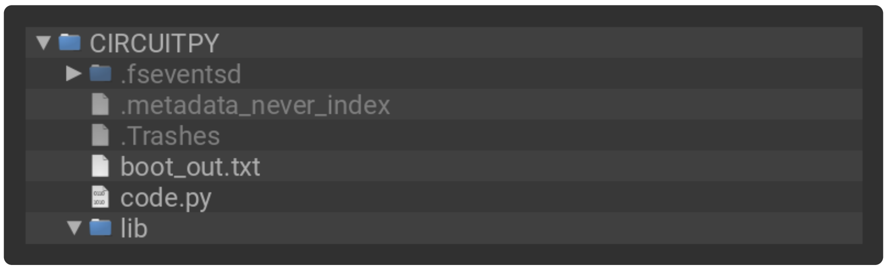
What Are All the Available Names?

Many pins on CircuitPython compatible microcontroller boards have multiple names, however, typically, there's only one name labeled on the physical board. So how do you find out what the other available pin names are? Simple, with the following script! Each line printed out to the serial console contains the set of names for a particular pin.

On a microcontroller board running CircuitPython, first, connect to the serial console.

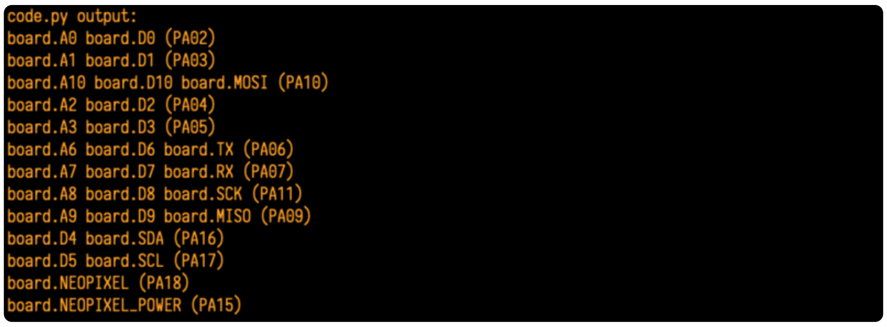
In the example below, click the Download Project Bundle button below to download the necessary libraries and the code.py file in a zip file. Extract the contents of the zip file, open the directory CircuitPython_Essentials/Pin_Map_Script/ and then click on the directory that matches the version of CircuitPython you're using and copy the contents of that directory to your CIRCUITPY drive.

Your CIRCUITPY drive should now look similar to the following image:



```
# SPDX-FileCopyrightText: 2020 anecdata for Adafruit Industries
# SPDX-FileCopyrightText: 2021 Neradoc for Adafruit Industries
# SPDX-FileCopyrightText: 2021-2023 Kattni Rembor for Adafruit Industries
# SPDX-FileCopyrightText: 2023 Dan Halbert for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""CircuitPython Essentials Pin Map Script"""
import microcontroller
import board
try:
    import cyw43  # raspberrypi
except ImportError:
    cyw43 = None

board_pins = []
for pin in dir(microcontroller.pin):
    if (isinstance(getattr(microcontroller.pin, pin), microcontroller.Pin) or
        (cyw43 and isinstance(getattr(microcontroller.pin, pin), cyw43.CywPin))):
        pins = []
        for alias in dir(board):
            if getattr(board, alias) is getattr(microcontroller.pin, pin):
                pins.append(f"board.{alias}")
        # Add the original GPIO name, in parentheses.
        if pins:
            # Only include pins that are in board.
            pins.append(f"({str(pin)})")
            board_pins.append(" ".join(pins))

for pins in sorted(board_pins):
    print(pins)
```

Here is the result when this script is run on QT Py SAMD21:



```
code.py output:
board.A0 board.D0 (PA02)
board.A1 board.D1 (PA03)
board.A10 board.D10 board.MOSI (PA10)
board.A2 board.D2 (PA04)
board.A3 board.D3 (PA05)
board.A6 board.D6 board.TX (PA06)
board.A7 board.D7 board.RX (PA07)
board.A8 board.D8 board.SCK (PA11)
board.A9 board.D9 board.MISO (PA09)
board.D4 board.SDA (PA16)
board.D5 board.SCL (PA17)
board.NEOPIXEL (PA18)
board.NEOPIXEL_POWER (PA15)
```

Each line represents a single pin. Find the line containing the pin name that's labeled on the physical board, and you'll find the other names available for that pin. For example, the first pin on the board is labeled A0. The first line in the output is `board.A0 board.D0 (PA02)`. This means that you can access pin A0 in CircuitPython using both `board.A0` and `board.D0`.

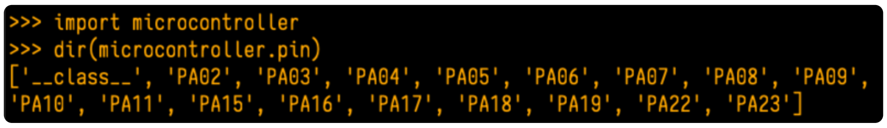
The pins in parentheses are the microcontroller pin names. See the next section for more info on those.

You'll notice there are two "pins" that aren't labeled on the board but appear in the list: `board.NEOPIXEL` and `board.NEOPIXEL_POWER`. Many boards have several of these special pins that give you access to built-in board hardware, such as an LED or an on-board sensor. The QT Py SAMD21 only has one on-board extra piece of hardware, a NeoPixel LED, so there's only the one available in the list. But you can also control whether or not power is applied to the NeoPixel, so there's a separate pin for that.

That's all there is to figuring out the available names for a pin on a compatible microcontroller board in CircuitPython!

Microcontroller Pin Names

The pin names available to you in the CircuitPython `board` module are not the same as the names of the pins on the microcontroller itself. The board pin names are aliases to the microcontroller pin names. If you look at the datasheet for your microcontroller, you'll likely find a pinout with a series of pin names, such as "PA18" or "GPIO5". If you want to get to the actual microcontroller pin name in CircuitPython, you'll need the `microcontroller.pin` module. As with `board`, you can run `dir(microcontroller.pin)` in the REPL to receive a list of the microcontroller pin names.



```
>>> import microcontroller
>>> dir(microcontroller.pin)
['__class__', 'PA02', 'PA03', 'PA04', 'PA05', 'PA06', 'PA07', 'PA08', 'PA09',
'PA10', 'PA11', 'PA15', 'PA16', 'PA17', 'PA18', 'PA19', 'PA22', 'PA23']
```

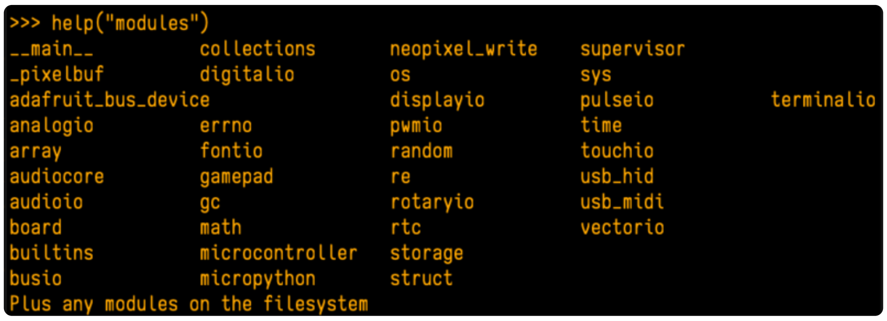
CircuitPython Built-In Modules

There is a set of modules used in most CircuitPython programs. One or more of these modules is always used in projects involving hardware. Often hardware requires installing a separate library from the Adafruit CircuitPython Bundle. But, if you try to find `board` or `digitalio` in the same bundle, you'll come up lacking. So, where do these modules come from? They're built into CircuitPython! You can find an comprehensive list of built-in CircuitPython modules and the technical details of their functionality from CircuitPython here () and the Python-like modules included here (). However, not every module is available for every board due to size constraints or hardware limitations. How do you find out what modules are available for your board?

There are two options for this. You can check the support matrix (), and search for your board by name. Or, you can use the REPL.

Plug in your board, connect to the serial console and enter the REPL. Type the following command.

```
help("modules")
```



```
>>> help("modules")
__main__          collections       neopixel_write    supervisor
_pixelbuf         digitalio         os                sys
adafruit_bus_device                 displayio         pulseio           terminalio
analogio          errno             pwmio             time
array             fontio            random            touchio
audiocore         gamepad           re                usb_hid
audioio           gc                rotaryio          usb_midi
board             math              rtc               vectorio
builtins          microcontroller   storage
busio             micropython       struct
Plus any modules on the filesystem
```

That's it! You now know two ways to find all of the modules built into CircuitPython for your compatible microcontroller board.

Blinky and a Button

To get you started with how to program your Pico in CircuitPython, especially for those who may have started out with the official MicroPython setup, we've 'ported' the Getting Started with MicroPython on Pico book () examples to CircuitPython. The book is awesome, please download/purchase it to support Raspberry Pi Press ()!

Printing `"Hello, World!"` is the traditional first program to write in any language. In CircuitPython, the Hello, World! equivalent is blinking an LED. This is easy to do with your Raspberry Pi Pico board and CircuitPython.

The Built-In LED

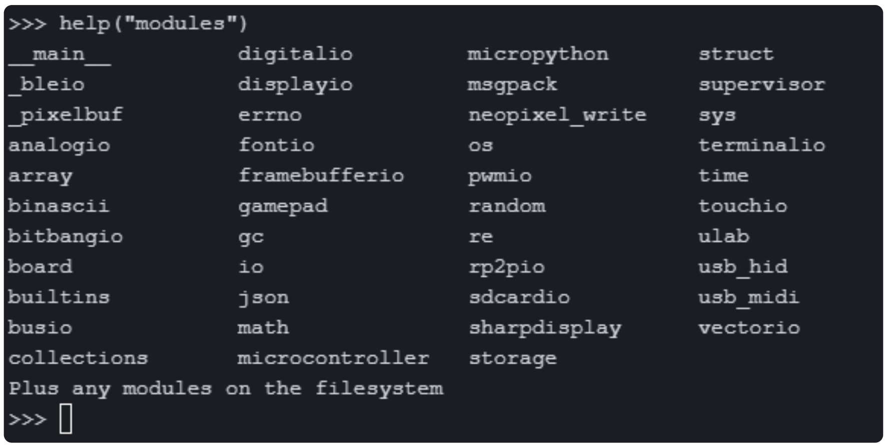
Your Pico board has a built in LED, labeled "LED", located to the left of the USB port at the top of the board. Like any other LED, it turns on when it is powered, and is otherwise off. The LED is connected to pin GP25. GP25 is dedicated to the LED, and therefore is not available as a pin along the edge of your Pico board. In CircuitPython, you can access this LED using `board.LED`.

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

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

"""Example for Pico. Turns on the built-in LED."""
import board
import digitalio

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

while True:
    led.value = True
```

So far, none of the examples have required importing anything. Most, if not all, hardware programming requires you to import built-in modules or libraries. CircuitPython libraries are not included in CircuitPython itself and require you to copy files or folders to the lib folder on your CIRCUITPY drive. The built-in modules, however, are included, and do not require any external files or folders. To see a list of available built-in modules for your Pico board, you can enter the REPL and type the following at the `>>>` prompt:

```
help("modules")
```



```
>>> help("modules")
__main__          digitalio         micropython       struct
_bleio            displayio         msgpack           supervisor
_pixelbuf         errno             neopixel_write    sys
analogio          fontio            os                terminalio
array             framebufferio     pwmio             time
binascii          gamepad           random            touchio
bitbangio         gc                re                ulab
board             io                rp2pio            usb_hid
builtins          json              sdcardio          usb_midi
busio             math              sharpdisplay      vectorio
collections       microcontroller   storage
Plus any modules on the filesystem
>>> 
```

This example requires two modules: `board` and `digitalio`.

The first step to hardware programming is identifying the location of the hardware board. The `board` module contains all of the pin names of the pins available on your Pico board. These are not the pin names on the chip itself! They are the names of the pins provided to CircuitPython for use in your code.

One of the most basic parts of interfacing with hardware is managing digital inputs and outputs. This is where `digitalio` comes in. The `digitalio` module allows you to digitally control IO pins, as well as set the direction and pull of the pin.

So, you `import` both `board` and `digitalio` at the beginning of the file:

```
import board
import digitalio
```

Next, you set up the LED. You assign the variable `led` to the `digitalio.DigitalInOut` object. This will allow you to manipulate the LED object using the `led` variable, instead of having to type out the entire object every time you want to use it. The `DigitalInOut` object takes one argument - the pin object using the `board` module, `board.LED`. Then you set the pin direction to `OUTPUT` to tell your Pico board that it should be used as an output (versus an input).

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

This setup code tells your Pico board how to talk to the LED. The next step is to tell it to turn on. To do this, you'll need to start a `while True:` loop. Within the loop, set the led value to True with `led.value = True`. Remember, for code to be "in" a loop, it must be indented under the `while True:`.

```
while True:
    led.value = True
```

The green LED turns on!

It's worth noting, if you simply set the led value to `True` without a loop, it would flash quickly once, and then remain turned off. This is due to the way CircuitPython works, with regard to what happens when your program ends. When your code finishes running, CircuitPython resets your Pico board to prepare it for the next run of code. That means the set up you did earlier no longer applies, and the LED does not remain turned on. To that end, most CircuitPython programs involve some kind of loop, infinite or otherwise.

You'll notice the LED stays on. This is because you have not told it to turn off. To turn the LED off, you set `led.value = False`. Try adding that line of code to the end of your current code.py file.

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

"""Example for Pico. Turns the built-in LED on and off with no delay."""
import board
import digitalio

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

while True:
    led.value = True
    led.value = False
```

The LED still doesn't appear to turn off. It is turning off! Your Pico board processes code so quickly that, to the naked eye, the LED does not appear to be turning off. However, it is rapidly flashing on and off, faster than you are able to process. The solution is to add a delay. For that, you need to import a new module called `time`. Then, you add a `time.sleep()` after turning the LED on, and after turning the LED off.

Update your code.py to the following and save.

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

"""Example for Pico. Blinks the built-in LED."""
import time
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 LED begins blinking!

Including a `sleep()` in your code tells the program to pause for the given number of seconds. In this case, it pauses for half of one second, or `0.5` seconds. Try changing `0.5` to `1` and see what happens! The two `sleep()` times do not have to be the same - try making them different to see the results.

There is a more concise but less clear way to do the same thing. Consider the following. Update your code.py to the following and save.

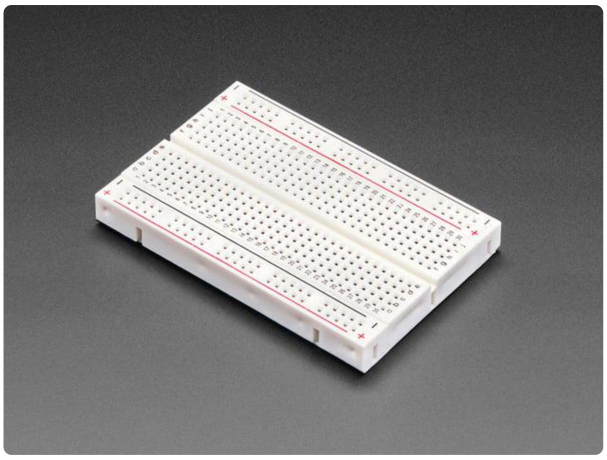
```
# SPDX-FileCopyrightText: 2021 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""Example for Pico. Blinks the built-in LED."""
import time
import board
import digitalio

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

while True:
    led.value = not led.value
    time.sleep(0.5)
```

The LED blinks!

Remember that when the LED value is `True`, it is on, and when it is `False`, it is off. In this example, the `not` is a logic operator that reverses the result. So, it turns a `True` into a `False`, and a `False` into a `True`. With the `0.5` second `sleep()`, this causes the LED value to cycle between `True` and `False` every `0.5` seconds. It does exactly the same thing as explicitly setting the value, but saves a couple of lines of code!

An External LED

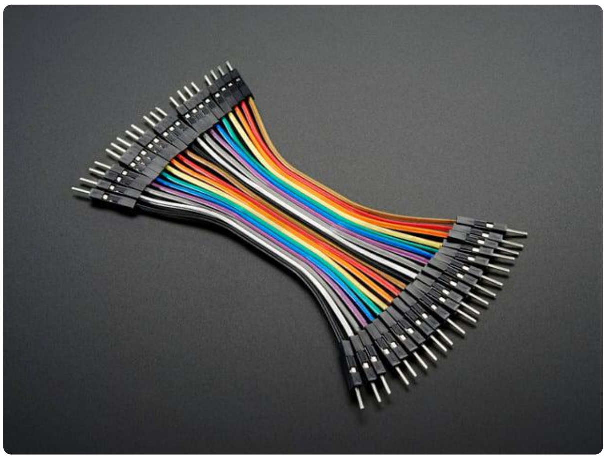
The first step to controlling an external LED is connecting one to your Pico board. For this example, you'll need your Pico board, a breadboard, male-to-male jumper wires, a resistor, and an LED. A 220Ω-1.0KΩ resistor will work; a 220Ω resistor is shown in the diagram.



Half Sized Premium Breadboard - 400 Tie Points

This is a cute, half-size breadboard with 400 tie points, good for small projects. It's 3.25" x 2.2" / 8.3cm x 5.5cm with a standard double-strip in the...

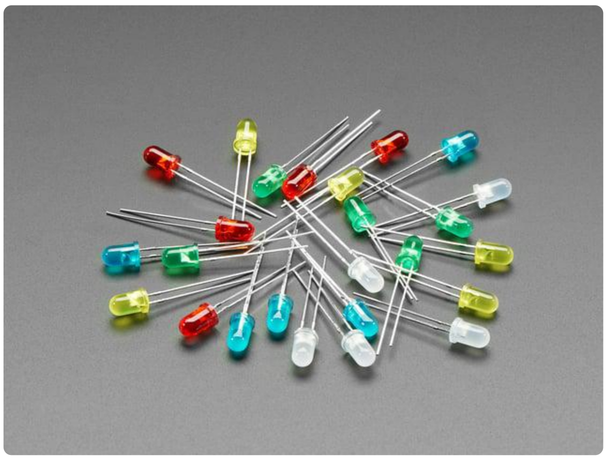
https://www.adafruit.com/product/64



Premium Male/Male Jumper Wires - 20 x 3" (75mm)

Handy for making wire harnesses or jumpering between headers on PCB's. These premium jumper wires are 3" (75mm) long and come in a 'strip' of 20 (2 pieces of each...

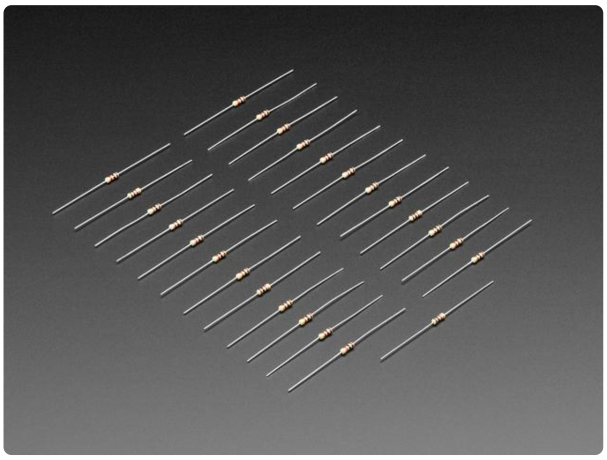
https://www.adafruit.com/product/1956



Diffused 5mm LED Pack - 5 LEDs each in 5 Colors - 25 Pack

Need some indicators? We are big fans of these diffused LEDs. They are fairly bright, so they can be seen in daytime, and from any angle. They go easily into a breadboard and will add...

https://www.adafruit.com/product/4203



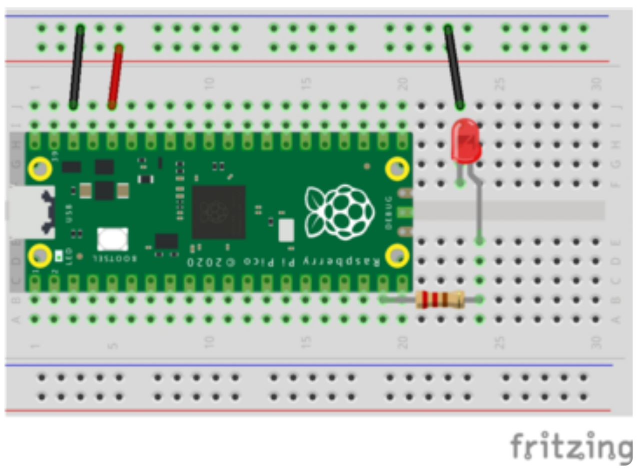
Through-Hole Resistors - 1.0K ohm 5% 1/4W - Pack of 25

ΩMG! You're not going to be able to resist these handy resistor packs! Well, axially, they do all of the resisting for you!This is a 25 Pack of...

https://www.adafruit.com/product/4294

Wire up the LED to your Pico board as shown below.

External LEDs must have a current-limiting resistor between the LED and your board. Without it, you can damage both the LED and your board!



Board GND to breadboard ground rail (using a jumper wire)
Board GP14 to 220**Ω** resistor
LED+ to 220**Ω** resistor
LED- to breadboard ground rail (using a jumper wire)

Now that you've wired up an LED to your Pico board, it's time to light it up. The code is almost exactly the same as the code used to turn on the built-in LED. The only change required is to update the pin provided in setup to the pin to which you connected your external LED. You connected the external LED to GP14. So, take the blink code from above, and change the pin to match the new pin assignment.

Update your code.py to the following and save.

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

"""
LED example for Pico. Blinks external LED on and off.

REQUIRED HARDWARE:
* LED on pin GP14.
"""
import time
import board
import digitalio

led = digitalio.DigitalInOut(board.GP14)
led.direction = digitalio.Direction.OUTPUT

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

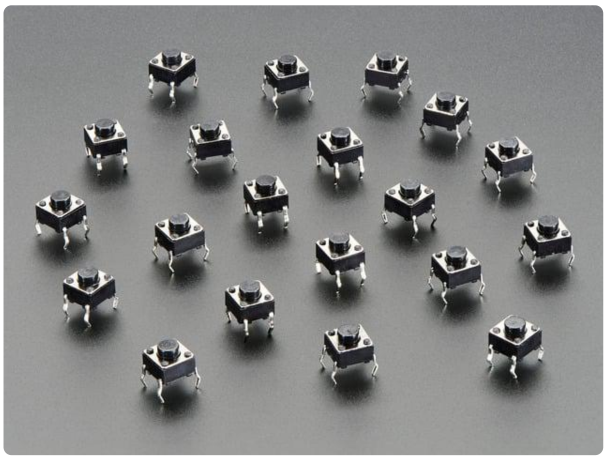
The external LED begins blinking! That's all it takes to basically control an external LED.

Now it's time to take a look at using hardware as an input (instead of an output like an LED).

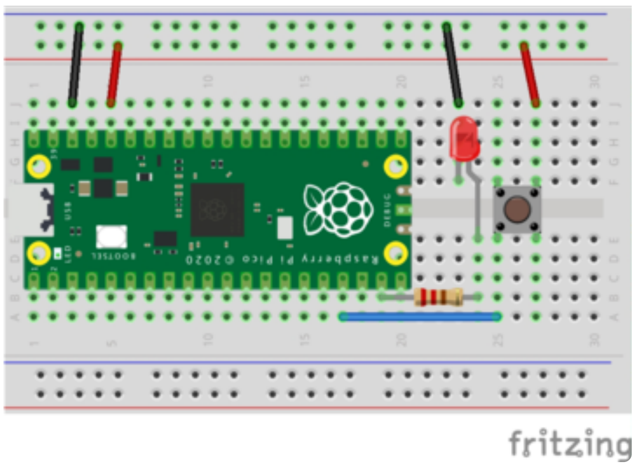
Using a Button as an Input

The IO in GPIO stands for input/output, which is to say that all GPIO pins can be used as both inputs and outputs. For this example, you'll need your current wiring setup, a button switch, and male-to-male jumper wires. The first step is connecting the button to your Pico board.

Wire up the button to your current setup as shown below. The button used in the diagram is a four-legged button. The legs are connected in pairs, so the simplest way to ensure you're wired up properly is to use the opposite legs. There are buttons that have only two legs, and in that case, you would still connect to the opposite legs.



Tactile Button switch (6mm) x 20 pack
Little clicky switches are standard input "buttons" on electronic projects. These work best in a PCB but
https://www.adafruit.com/product/367



Board 3V3 to breadboard power rail (using jumper wire)
Board GP13 to leg of button (using jumper wire)
Opposite leg of button to breadboard power rail (using jumper wire)

Now that you have wired up your button, it's time to read status from it. Setup will look a lot like setting up the LED, with a couple of important differences.

Update your code.py to the following and save.

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

"""
Button example for Pico. Prints button pressed state to serial console.

REQUIRED HARDWARE:
* Button switch on pin GP13.
"""
import time
import board
import digitalio

button = digitalio.DigitalInOut(board.GP13)
button.switch_to_input(pull=digitalio.Pull.DOWN)

while True:
    print(button.value)
    time.sleep(0.5)
```

You import the same three modules: `time`, `board` and `digitalio`.

```
import time
import board
import digitalio
```

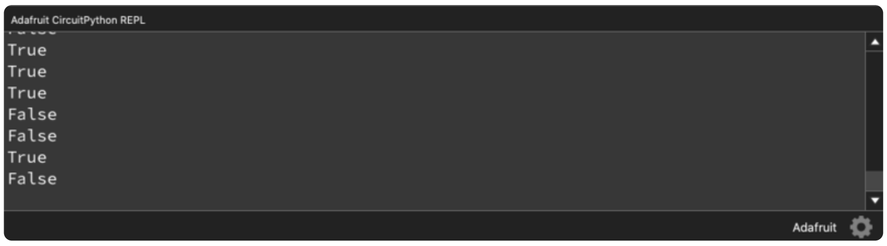
Then, as with the LED, you assign the variable `button` to a `digitalio.DigitalInOut` object, and provide it the pin you used to connect it to your Pico board - GP13. Then, unlike the LED, you set it to an input, and you set the pull to DOWN.

```
button = digitalio.DigitalInOut(board.GP13)
button.switch_to_input(pull=digitalio.Pull.DOWN)
```

This type of button is called a momentary button switch. When the button is not pressed, the opposite legs are not connected, and when you press the button, it makes a connection between the opposite legs. Pulling the pin down registers 0V when it is not connected to anything, so connecting it to 3.3V, e.g. pressing the button, registers a button press.

To check whether or not the button is pressed, you'll `print` the `button.value` in a loop. The `sleep()` is included to keep the serial output readable - without a delay, the serial output would be incredibly fast!

```
while True:
    print(button.value)
    time.sleep(0.5)
```



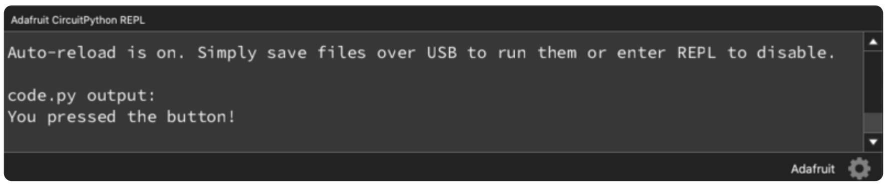
```
True
True
True
False
False
True
False
```

Perhaps you would rather print a message only when the button is pressed.

Update your code.py to the following and save.

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

"""
Button example for Pico. Prints message to serial console when button is pressed.

REQUIRED HARDWARE:
* Button switch on pin GP13.
"""
import time
import board
import digitalio

button = digitalio.DigitalInOut(board.GP13)
button.switch_to_input(pull=digitalio.Pull.DOWN)

while True:
    if button.value:
        print("You pressed the button!")
        time.sleep(0.5)
```

Now, check the serial console. Nothing is happening because you a have not pressed the button. Try pressing the button.



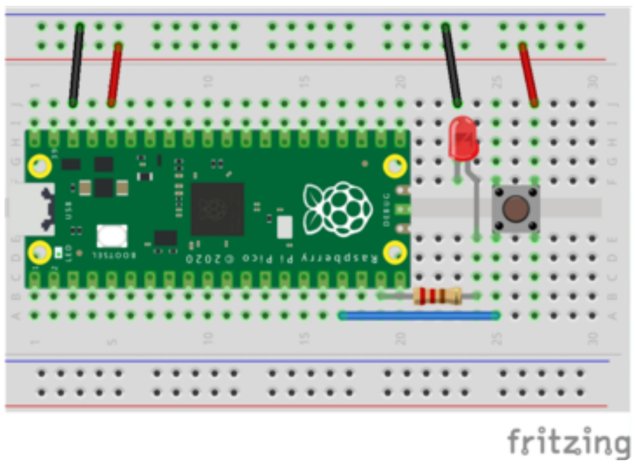
```
Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.

code.py output:
You pressed the button!
```

If you continue to press the button, it will display the message every 0.5 seconds until you let go. Now it's time to mix things up!

Control an External LED with a Button

Most electronics examples involve multiple components, which is why your Pico board has so many GPIO pins on it. You've learned about controlling an external LED, and reading the input from a button. You can combine those two concepts and control the LED using the button!

Now you understand why you left the LED connected to your Pico board when the previous example didn't involve it. You've already connected everything needed for this example!



Your hardware setup should still look like this.

Update your code.py to the following and save.

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

"""
Button and LED example for Pico. Turns on LED when button is pressed.

REQUIRED HARDWARE:
* Button switch on pin GP13.
* LED on pin GP14.
"""
import board
import digitalio

led = digitalio.DigitalInOut(board.GP14)
led.direction = digitalio.Direction.OUTPUT
button = digitalio.DigitalInOut(board.GP13)
button.switch_to_input(pull=digitalio.Pull.DOWN)

while True:
    if button.value:
        led.value = True
    led.value = False
```

Press the button. The LED turns on! Let it go, and the LED turns off.

You only need to import two modules this time: `board` and `digitalio`.

```
import board
import digitalio
```

Setup is the same for the LED and the button as it was previous, however, this time you include both.

```
led = digitalio.DigitalInOut(board.GP14)
led.direction = digitalio.Direction.OUTPUT
button = digitalio.DigitalInOut(board.GP13)
button.switch_to_input(pull=digitalio.Pull.DOWN)
```

Finally, in your loop, you check to see if the button is pressed, and if so, you turn on the LED. Otherwise, you turn off the LED.

```
while True:
    if button.value:
        led.value = True
    led.value = False
```

Alternatively, you can simply set the LED value equal to the button value, and you get the same results. Remember, when the button is pressed, it returns `True`, and when it's not, it returns `False`. When the LED is set to `True`, it turns on, and when it's set to `False`, it turns off. It's a quick way to control an LED with a button press!

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

"""
Button and LED example for Pico. Turns on LED when button is pressed.

REQUIRED HARDWARE:
* Button switch on pin GP13.
* LED on pin GP14.
"""
import board
import digitalio

led = digitalio.DigitalInOut(board.GP14)
led.direction = digitalio.Direction.OUTPUT
button = digitalio.DigitalInOut(board.GP13)
button.switch_to_input(pull=digitalio.Pull.DOWN)

while True:
    led.value = button.value
```

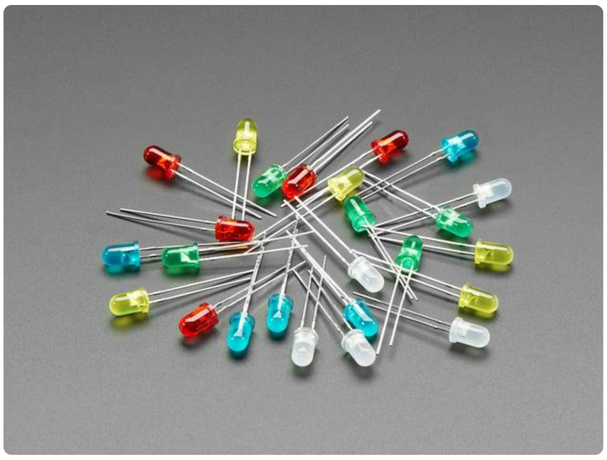
Traffic Light and Pedestrian Crossing

To get you started with how to program your Pico in CircuitPython, especially for those who may have started out with the official MicroPython setup, we've 'ported' the Getting Started with MicroPython on Pico book () examples to CircuitPython. The book is awesome, please download/purchase it to support Raspberry Pi Press ()!

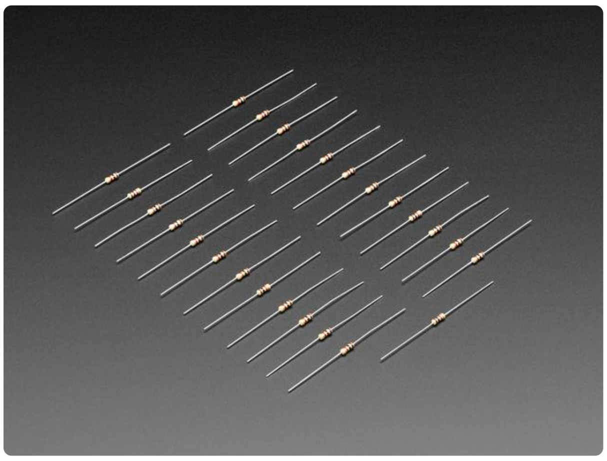
Traffic lights and pedestrian crossings are something most folks encounter on a regular basis, but perhaps never give much consideration, especially to the fact that they use microcontrollers. At their simplest, each of the light colors, red, amber and green, turn on and off for a predetermined period of time. More typically, they are far more complicated, involving things like inter-system communication, and tracking traffic flow and pedestrian crossings. While building a typical traffic management system is an incredibly advanced project, it's easy to build a simple simulator using a microcontroller and a few components.

This example will show you how to use your Raspberry Pi Pico board to control multiple LEDs using different timings, and how to monitor for a button press while other code is running using polling.

Parts Used



Diffused 5mm LED Pack - 5 LEDs each in 5 Colors - 25 Pack
Need some indicators? We are big fans of these diffused LEDs. They are fairly bright, so they can be seen in daytime, and from any angle. They go easily into a breadboard and will add...
https://www.adafruit.com/product/4203

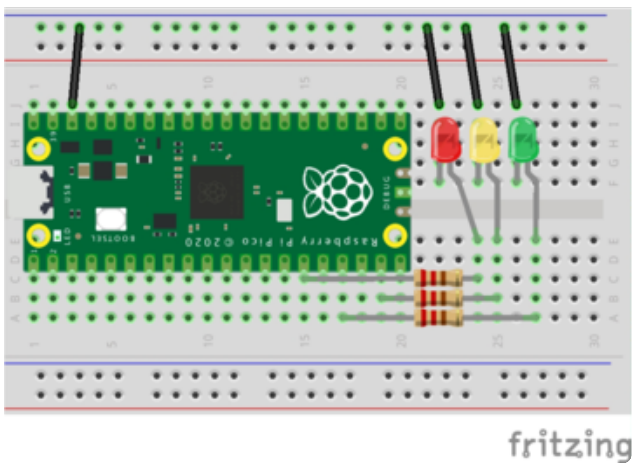


Through-Hole Resistors - 1.0K ohm 5% 1/4W - Pack of 25
ΩMG! You're not going to be able to resist these handy resistor packs! Well, axially, they do all of the resisting for you!This is a 25 Pack of...
https://www.adafruit.com/product/4294

Wiring the Traffic Light

For this example, you'll need your Pico board, a red LED, an amber LED and a green LED, three resistors, and a number of male-to-male jumper wires. A 220Ω-1.0KΩ resistor will work; a 220Ω resistor is shown in the diagram.

The first step is to wire up the LEDs. Connect the LEDs to your Pico board as shown below.



Board GND to breadboard ground rail (using a jumper wire)
Board GP11 to 220Ω resistor
Red LED+ to 220Ω resistor (connected to GP11)
Red LED- to breadboard ground rail (using a jumper wire)
Board GP14 to 220Ω resistor
Amber LED+ to 220Ω resistor (connected to GP14)
Amber LED- to breadboard ground rail (using a jumper wire)
Board GP13 to 220Ω resistor
Green LED+ to 220Ω resistor (connected to GP13)
Green LED- to breadboard ground rail (using a jumper wire)

Programming the Traffic Light

Now that you've wired up your traffic light, you can begin programming it.

Update your code.py to the following and save.

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

"""Traffic light simulator example for Pico. Turns on red, amber and green LEDs in 
traffic
light-like sequence.

REQUIRED HARDWARE:
* Red LED on pin GP11.
* Amber LED on pin GP14.
* Green LED on pin GP13.
"""
import time
import board
import digitalio

red_led = digitalio.DigitalInOut(board.GP11)
red_led.direction = digitalio.Direction.OUTPUT
amber_led = digitalio.DigitalInOut(board.GP14)
amber_led.direction = digitalio.Direction.OUTPUT
green_led = digitalio.DigitalInOut(board.GP13)
green_led.direction = digitalio.Direction.OUTPUT

while True:
    red_led.value = True
    time.sleep(5)
    amber_led.value = True
    time.sleep(2)
    red_led.value = False
    amber_led.value = False
    green_led.value = True
    time.sleep(5)
    green_led.value = False
    amber_led.value = True
    time.sleep(3)
    amber_led.value = False
```

The red LED lights up, followed by the amber LED. Both turn off as the green LED lights up. Green turns off, and amber lights up. Amber turns off, and the cycle begins again with the red LED.

Now, a detailed look at the code. First, you import the necessary modules.

```
import time
import board
import digitalio
```

Next, you set up the LEDs. As with the single red LED, each of the three LEDs requires assigning a variable to a `digitalio` object, and setting the pin direction to output.

```
red_led = digitalio.DigitalInOut(board.GP11)
red_led.direction = digitalio.Direction.OUTPUT
amber_led = digitalio.DigitalInOut(board.GP14)
amber_led.direction = digitalio.Direction.OUTPUT
green_led = digitalio.DigitalInOut(board.GP13)
green_led.direction = digitalio.Direction.OUTPUT
```

Variables should be descriptive and make it simple to figure out what they apply to. Previously, with only one LED, it was reasonable to call the variable `led`. Since you are now working with three different colors of LEDs, you should be more specific with your variables, so you know what you're working with in your code. Therefore, the variable names were updated to `red_led`, `amber_led`, and `green_led`.

Next you begin an infinite loop, and use `time.sleep()` to control the timing of the LEDs.

```
while True:
    red_led.value = True
    time.sleep(5)
    amber_led.value = True
    time.sleep(2)
    red_led.value = False
    amber_led.value = False
    green_led.value = True
    time.sleep(5)
    green_led.value = False
    amber_led.value = True
    time.sleep(3)
    amber_led.value = False
```

This timing sequence is based on actual traffic lights in the United Kingdom, shortened significantly for this example. Five seconds is hardly enough time for actual traffic flow!

First, you turn on the red LED, and wait 5 seconds. Then you turn on the amber LED for 2 seconds, to signal the light is about to change. The red LED stays on because you have not yet told it to turn off. Next, you turn off both the red and amber LEDs. Then you turn on the green LED for 5 seconds, followed by turning off the green LED. Finally you turn on the amber LED for 3 seconds, and then turn it off. At that point, the red LED turns on again because the loop has completed and restarted.

This example allows for traffic, but doesn't take pedestrians into account. The next example does exactly that.

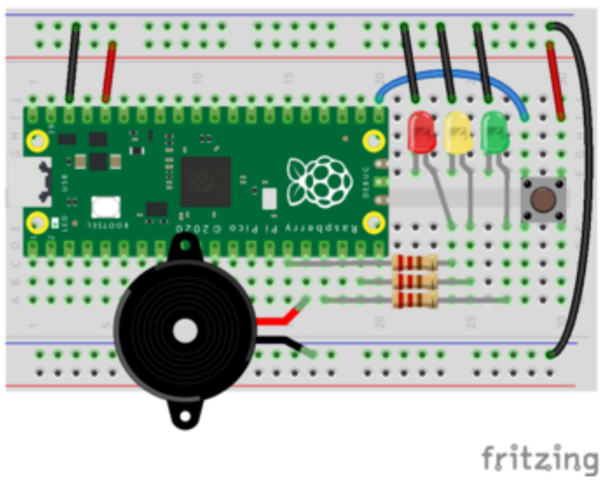
Traffic Light and Pedestrian Crossing

In the real world, traffic lights are designed not only for road vehicles, but also to allow pedestrians to safely cross streets. Now, it's time to take a pedestrian into consideration. Adding a couple of components can turn your traffic light into a pedestrian crossing.

Wiring the Pedestrian Crossing

This example requires your current hardware setup, plus a button switch, a piezo buzzer, and a few more male-to-male jumper wires.

Connect the button switch and piezo to your current setup as shown below. The button is shown with jumper wires connected to two legs on the same side. Each pair of legs is shaped a bit like a staple. You should be able to identify the separate pairs and ensure you're connected to the opposite legs, even though you have the wires on the same side. The direction of the piezo doesn't matter.



Board 3V3 to breadboard power rail (using jumper wire)
Board GP16 to leg of button (using jumper wire)
Opposite leg of button to breadboard power rail (using jumper wire)
Board GP12 to leg of piezo buzzer (using jumper wire)
Other leg of piezo buzzer to breadboard ground rail (using jumper wire)

Programming the Traffic Light and Pedestrian Crossing

Update your code.py to the following and save.

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

"""Traffic light with pedestrian crossing simulator example for Pico. Turns on red,
amber and green
LEDs in traffic light-like sequence. When button is pressed, upon light sequence
completion, the
red LED turns on and the buzzer beeps to indicate pedestrian crossing is active.

REQUIRED HARDWARE:
* Red LED on pin GP11.
* Amber LED on pin GP14.
* Green LED on pin GP13.
* Button switch on pin GP16.
* Piezo buzzer on pin GP13.
"""
import time
import board
import digitalio
import pwmio

red_led = digitalio.DigitalInOut(board.GP11)
red_led.direction = digitalio.Direction.OUTPUT
amber_led = digitalio.DigitalInOut(board.GP14)
amber_led.direction = digitalio.Direction.OUTPUT
green_led = digitalio.DigitalInOut(board.GP13)
green_led.direction = digitalio.Direction.OUTPUT
button = digitalio.DigitalInOut(board.GP16)
button.switch_to_input(pull=digitalio.Pull.DOWN)
buzzer = pwmio.PWMOut(board.GP12, frequency=660, duty_cycle=0,
variable_frequency=True)

button_pressed = False


def waiting_for_button(duration):
    global button_pressed  # pylint: disable=global-statement
    end = time.monotonic() + duration
    while time.monotonic() < end:
        if button.value:
            button_pressed = True
```

```
while True:
    if button_pressed:
        red_led.value = True
        for _ in range(10):
            buzzer.duty_cycle = 2 ** 15
            waiting_for_button(0.2)
            buzzer.duty_cycle = 0
            waiting_for_button(0.2)
        button_pressed = False
    red_led.value = True
    waiting_for_button(5)
    amber_led.value = True
    waiting_for_button(2)
    red_led.value = False
    amber_led.value = False
    green_led.value = True
    waiting_for_button(5)
    green_led.value = False
    amber_led.value = True
    waiting_for_button(3)
    amber_led.value = False
```

The light sequence is the same as before. This time, however, you can press the button, and when the light sequence completes, the red LED will stay on and the piezo will play a series of beeps. Then the light sequence will begin again. You can repeat this as often as you like.

Now, a detailed look at the code. First, you import the necessary modules. This example uses the same three as the previous example, but also requires a new module: `pwmio`.

```
import time
import board
import digitalio
import pwmio
```

Then you set up all of the hardware components. The LED setup is the same as before, but you are adding in the button and the buzzer.

```
red_led = digitalio.DigitalInOut(board.GP11)
red_led.direction = digitalio.Direction.OUTPUT
amber_led = digitalio.DigitalInOut(board.GP14)
amber_led.direction = digitalio.Direction.OUTPUT
green_led = digitalio.DigitalInOut(board.GP13)
green_led.direction = digitalio.Direction.OUTPUT
button = digitalio.DigitalInOut(board.GP16)
button.switch_to_input(pull=digitalio.Pull.DOWN)
buzzer = pwmio.PWMOut(board.GP12, frequency=660, duty_cycle=0,
variable_frequency=True)
```

The button is set up as it as in previous examples, however the pin it is connected to on your Pico board has changed, so make sure you use the current pin.

The buzzer set up is different than any previous examples. You can use PWM with variable frequency to play tones using piezo buzzers. To use PWM with CircuitPython, you use the `pwmio` module. You create the buzzer variable and assign it to a `pwmio.PWMOut` object. This object can take up to four arguments: `pin`, `frequency`, `duty_cycle` and `variable_frequency`. To play tones, you'll need to provide all four. `pin` is the pin to which you wired the piezo. `frequency` is the frequency of the tone - in this case 660Hz. `duty_cycle` is set to `0` when the object is created, or your piezo will play the tone constantly as long as a loop is present in your code. Finally, `variable_frequency` is set to `True` to allow for changing the frequency later in the code.

Then, you create a variable called `button_pressed` and set it to False initially, as the button is not initially pressed.

```
button_pressed = False
```

In this example, you need a way to monitor for button presses while controlling the timing of events in the code, such as lighting up and turning off the LEDs. In CircuitPython, this is done using polling, wherein the program is repeatedly reading a value until it changes.

In all the previous examples, you've used `time.sleep()` to control the timing of events. This is an excellent method for many use cases. However, during `sleep()`, the code is essentially paused. Therefore, the board cannot accept any other inputs or perform any other functions for that period of time. This type of code is referred to as being blocking. In many cases, this is not an issue, but in this case, you are waiting for a button to be pressed, so you'll need to do things a little differently.

This is where another function of the `time` module comes in: `monotonic()`. At any given point in time, `time.monotonic()` is equal to the number seconds since your board was last power-cycled. (The soft-reboot that occurs with the auto-reload when you save changes to your CircuitPython code, or enter and exit the REPL, does not start it over.) When it is called, it returns a number with a decimal, which is called a float. If, for example, you assign `time.monotonic()` to a variable, and then call `monotonic()` again to assign into a different variable, each variable is equal to the number of seconds that `time.monotonic()` was equal to at the time the variables were assigned. You can then subtract the first variable from the second to obtain the amount of time that passed. Here is a simple example ().

For this example, you'll create a `waiting_for_button()` function that will "act as" the `sleep()` function did in previous examples.

```
def waiting_for_button(duration):
    global button_pressed  # pylint: disable=global-statement
    end = time.monotonic() + duration
    while time.monotonic() &lt; end:
        if button.value:
            button_pressed = True
```

The `waiting_for_button()` function requires a duration in seconds. Variables created outside of functions are considered global variables. However, if you want to change a global variable within a function, you need to use the `global` keyword. As this function changes the state of the `button_pressed` variable, the first thing within the function is `global button_pressed`. Next, you create the `end` variable and assign it to `time.monotonic() + duration` which is the current time plus the provided duration. Then, you create a loop that checks the current value of `time.monotonic()` and checks to see if it is less than the `end` time. As long as that is valid, it looks for the button to be pressed, in which case `button.value` is True, and sets the `button_pressed` variable to `True`. Because a `duration` in seconds is provided every time this function is called, the code spends that duration checking for a button press. As the code is able to register a button press at any other time as well, the code is no longer blocking, and is able to register a button press at any time!

The last part of the code is an infinite loop. The first thing inside the loop is an `if` block that is looking for the button press.

```
while True:
    if button_pressed:
        red_led.value = True
        for _ in range(10):
            buzzer.duty_cycle = 2 ** 15
            waiting_for_button(0.2)
            buzzer.duty_cycle = 0
            waiting_for_button(0.2)
        button_pressed = False
```

The `if` block checks to see if the `button_pressed` variable is `True`, and if it is, executes the code within the block. If it is `True`, it turns on the red LED. Then, the `for _ in range(10):` tells the code to loop 10 times over the four lines of code indented below it. This section sets the `duty_cycle` of the piezo buzzer to half which causes it to play a tone, pauses for `0.2` seconds, then sets the `duty_cycle` to 0 to stop playing the tone, and pauses again for `0.2` seconds. This effectively causes a series of 10 beeps. And finally, it resets the `button_pressed` variable to `False`, so the code is ready again to wait for another button press.

The last part of the infinite loop looks similar to the simple traffic light example, but instead of using `time.sleep()`, it is using the `waiting_for_button()` function you created earlier in this example.

```
[...]
    red_led.value = True
    waiting_for_button(5)
    amber_led.value = True
    waiting_for_button(2)
    red_led.value = False
    amber_led.value = False
    green_led.value = True
    waiting_for_button(5)
    green_led.value = False
    amber_led.value = True
    waiting_for_button(3)
    amber_led.value = False
```

The timing and LED color sequence are the same. Now, press the button. If the program is currently in the middle of the loop, nothing will happen until it reaches the end of the loop and begins again. When the loop starts over following a button press, the LED will turn red, and the buzzer will beep, indicating it is safe for the pedestrian to cross! After that section of code is completed, the LED sequence will begin again with the red LED lighting up for 5 seconds. This is similar to how actual pedestrian crossings work - the light remains red after the crossing indicates it is no longer safe to ensure folks in the middle of crossing make it to the other side safely.

Press the button again to begin the sequence again. You can do this as many times as you like, the code will continue to run. That's what goes into creating a traffic light with pedestrian crossing!

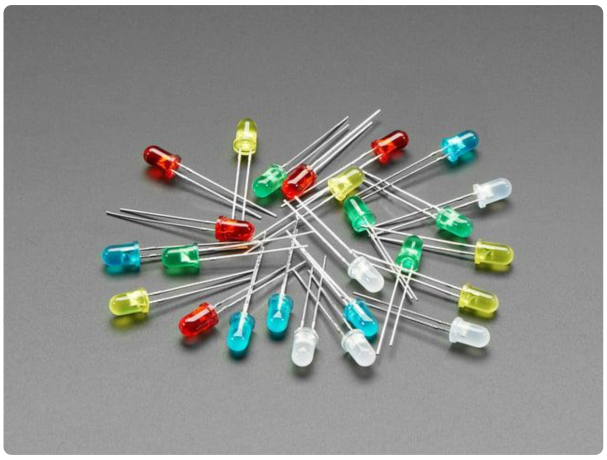
Reaction Game

To get you started with how to program your Pico in CircuitPython, especially for those who may have started out with the official MicroPython setup, we've 'ported' the Getting Started with MicroPython on Pico book () examples to CircuitPython. The book is awesome, please download/purchase it to support Raspberry Pi Press ()!

Studying human reaction time is literally a science (). There are many ways to test reaction time, but turning it into a game can make it fun. It's super easy to build a simple reaction game using a microcontroller, a few components and CircuitPython.

In this section, you'll learn how to use your Raspberry Pi Pico board to build a single-player reaction game, including an LED and a button, that tells you your reaction time in milliseconds. Then, add a more competitive aspect to it with the two-player version, built on the first example and including a second button, that tells you who pressed their button first.

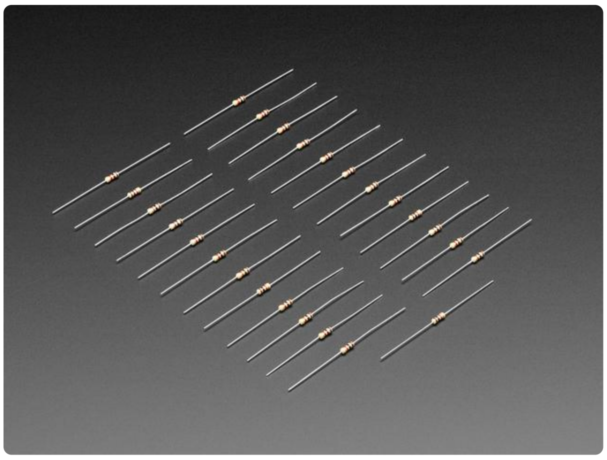
Parts Used



Diffused 5mm LED Pack - 5 LEDs each in 5 Colors - 25 Pack

Need some indicators? We are big fans of these diffused LEDs. They are fairly bright, so they can be seen in daytime, and from any angle. They go easily into a breadboard and will add...

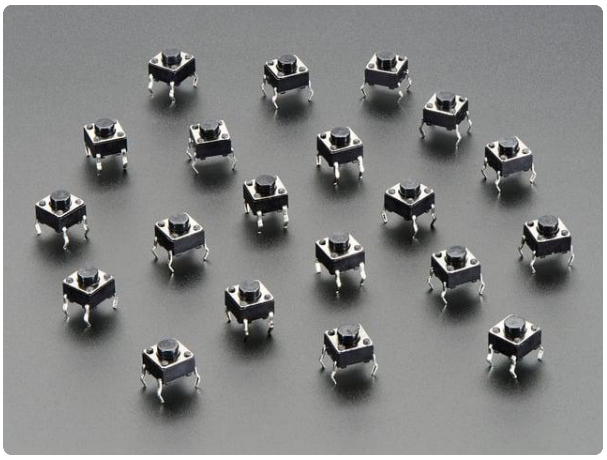
https://www.adafruit.com/product/4203



Through-Hole Resistors - 1.0K ohm 5% 1/4W - Pack of 25

ΩMG! You're not going to be able to resist these handy resistor packs! Well, axially, they do all of the resisting for you!This is a 25 Pack of...

https://www.adafruit.com/product/4294



Tactile Button switch (6mm) x 20 pack

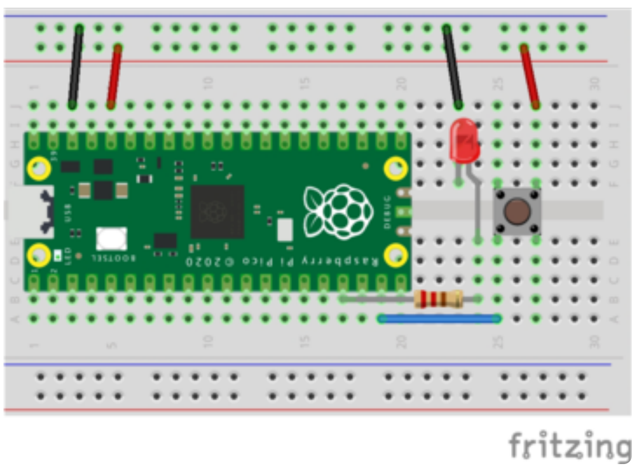
Little clicky switches are standard input "buttons" on electronic projects. These work best in a PCB but

https://www.adafruit.com/product/367

Wiring the Reaction Game

For this example you'll need your Pico board, an LED, a resistor, a button switch, and a number of male to male jumper wires. A 220Ω-1.0KΩ resistor will work; a 220Ω resistor is shown in the diagram.

The first step is to connect the LED and button to your Pico board. The diagram uses a red LED, but you can choose any color, Wire them up as shown below.



Board GND to breadboard ground rail (using a jumper wire)
Board 3V3 to breadboard power rail (using jumper wire)
Board GP13 to 220**Ω** resistor
LED+ to 220**Ω** resistor
LED- to breadboard ground rail (using a jumper wire)
Board GP14 to leg of button (using a jumper wire)
Opposite leg of button to breadboard power rail (using jumper wire)

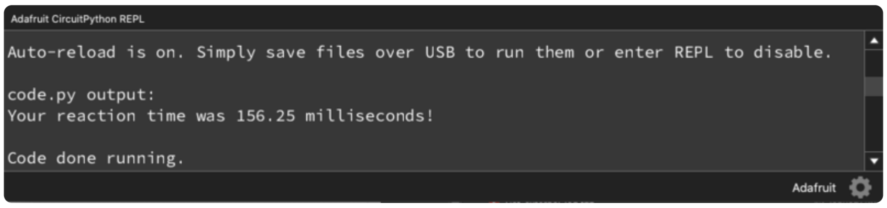
Programming the Reaction Game

Now that you've wired up your reaction game, you can begin programming it.

Update your code.py to the following and save.

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

"""
Reaction game example for Pico. LED turns on for between 5 and 10 seconds. Once it
turns off, try
to press the button as quickly as possible to measure reaction timm.

REQUIRED HARDWARE:
* LED on pin GP13.
* Button switch on pin GP14.
"""
import time
import random
import board
import digitalio

led = digitalio.DigitalInOut(board.GP13)
led.direction = digitalio.Direction.OUTPUT
button = digitalio.DigitalInOut(board.GP14)
button.switch_to_input(pull=digitalio.Pull.DOWN)

led.value = True
time.sleep(random.randint(5, 10))
led.value = False
timer_start = time.monotonic()
while True:
    if button.value:
        reaction_time = (time.monotonic() - timer_start) * 1000  # Convert to ms
        print("Your reaction time was", reaction_time, "milliseconds!")
        break
```

The LED will turn on for a period of time. Once it turns off, try to press the button as quickly as you can. Your reaction time will show up in the serial console! To restart the game, click in the serial console and press CTRL+D to reload the board.



```
Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.

code.py output:
Your reaction time was 156.25 milliseconds!

Code done running.
```

Now, a more detailed look at the code. First you import the necessary modules.

```
import time
import random
import board
import digitalio
```

You've imported `time`, `board` and `digitalio` previously, but this example uses a new module as well: `random`.

Next, you set up the LED and the button.

```
led = digitalio.DigitalInOut(board.GP13)
led.direction = digitalio.Direction.OUTPUT
button = digitalio.DigitalInOut(board.GP14)
button.switch_to_input(pull=digitalio.Pull.DOWN)
```

Then, you begin the game. First, you turn on the LED. Then you use a `time.sleep()` to keep it on for a random amount of time, between 5 and 10 seconds. The `random` module includes the `randint` function, which returns a random integer between the two provided integers, in this case `5` and `10`. Provide that function call to `time.sleep()`, and it will use the integer as seconds. This is followed by you turning off the LED.

```
led.value = True
time.sleep(random.randint(5, 10))
led.value = False
```

Then you begin the timer by setting a variable, `timer_start`, equal to `time.monotonic()`.

```
timer_start = time.monotonic()
```

Now, you create a loop to check for the button press. Once the button is pressed, the code checks the current value of `time.monotonic()` and subtracts from that the initial value, e.g. when the timer started. This value is multiplied by 1000 to convert the value from seconds to milliseconds. Then, the message, `Your reaction time was ### milliseconds!` is printed to the serial console, where `###` is the number of milliseconds between the time the LED turned off and the time you pressed the button.

```
while True:
    if button.value:
        reaction_time = (time.monotonic() - timer_start) * 1000  # Convert to ms
        print("Your reaction time was", reaction_time, "milliseconds!")
        break
```

The `break` is included because without it, the loop would continue to repeat following the button press, and the message would be spammed to the serial console repeatedly. You want to run the calculation once, and print the message once. The `break` stops the loop from continuing, and the code stops running following the button press.

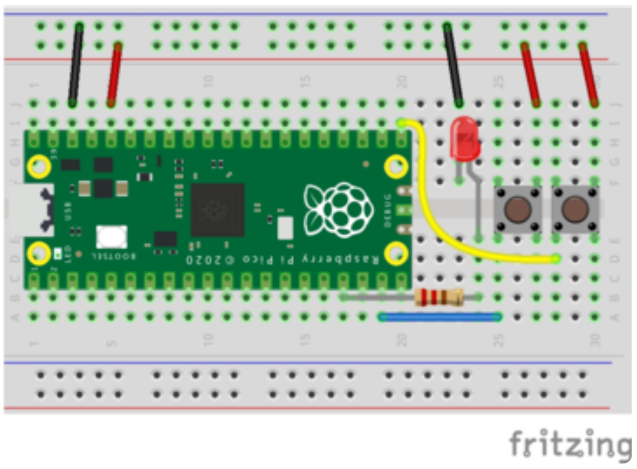
Two Players Makes It More Fun

Trying to best yourself in a reaction game can be fun, but trying to best someone else can be even better. You can easily involve them in the game above; invite them to play and then compare reaction times to see who is fastest. But, with a few modifications to the hardware and software, you can create a game you can play together!

Wiring the Two Player Reaction Game

For this example, you'll need your current hardware setup, another button switch and a few more male-to-male jumper wires.

Add the second button switch as shown below.



Board GP16 to leg of second button (using a jumper wire)
Opposite leg of second button to breadboard power rail (using jumper wire)

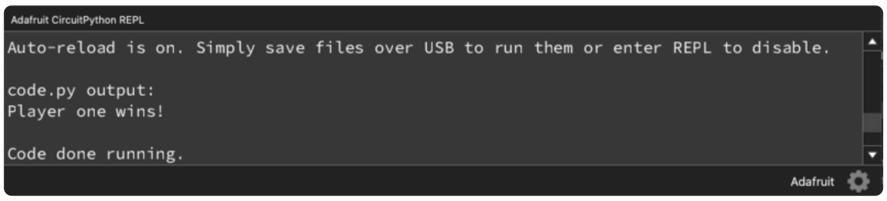
Programming the Two Player Reaction Game

This example looks quite similar to the single-player version with a few modifications.

Update your code.py to the following and save.

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

"""
Two-player reaction game example for Pico. LED turns on for between 5 and 10
seconds. Once it
turns off, try to press the button faster than the other player to see who wins.

REQUIRED HARDWARE:
* LED on pin GP13.
* Button switch on pin GP14.
* Button switch on GP16.
"""
import time
import random
import board
import digitalio

led = digitalio.DigitalInOut(board.GP13)
led.direction = digitalio.Direction.OUTPUT
button_one = digitalio.DigitalInOut(board.GP14)
button_one.switch_to_input(pull=digitalio.Pull.DOWN)
button_two = digitalio.DigitalInOut(board.GP16)
button_two.switch_to_input(pull=digitalio.Pull.DOWN)

led.value = True
time.sleep(random.randint(5, 10))
led.value = False
while True:
    if button_one.value:
        print("Player one wins!")
        break
    if button_two.value:
        print("Player two wins!")
        break
```

The LED still turns on for a period of time. Once it turns off, both of you can press your respective buttons. The winner is mentioned in a message printed to the serial console! To restart the game, click in the serial console and press CTRL+D to reload the board.



```
Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.

code.py output:
Player one wins!

Code done running.
```

Now, a detailed look at the code. The imports are identical to the single-player version. The first modification needed is to include the setup for the second button. As there are now two buttons, you'll need to update the variable name for the existing button as well. Setup now looks like the following.

```
led = digitalio.DigitalInOut(board.GP13)
led.direction = digitalio.Direction.OUTPUT
button_one = digitalio.DigitalInOut(board.GP14)
button_one.switch_to_input(pull=digitalio.Pull.DOWN)
button_two = digitalio.DigitalInOut(board.GP16)
button_two.switch_to_input(pull=digitalio.Pull.DOWN)
```

The code that turns the LED on for a random number of seconds between 5 and 10 remains the same.

```
led.value = True
time.sleep(random.randint(5, 10))
led.value = False
```

This time, however, you do not start a timer as you are not recording a reaction time, you are checking for which button is pressed first. So, the loop is significantly different. This time, you're repeatedly checking for a button press, possible from both button one and button two.

```
while True:
    if button_one.value:
        print("Player one wins!")
        break
    if button_two.value:
        print("Player two wins!")
        break
```

Based on whichever button is pressed first, the appropriate message prints to the serial console, and the code stops the loop.

Click in the serial console and press CTRL+D to restart the game, and play again!

Burglar Alarm

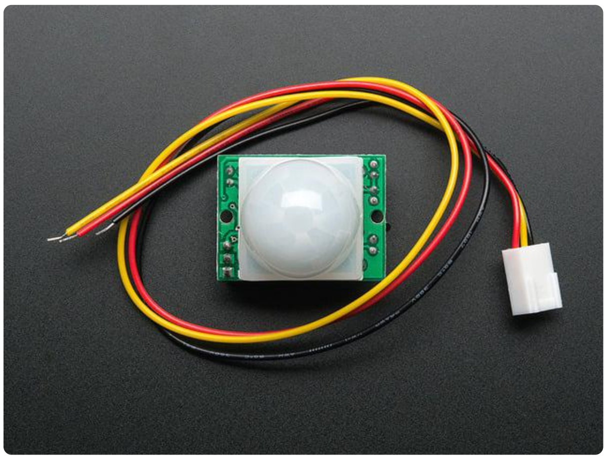
To get you started with how to program your Pico in CircuitPython, especially for those who may have started out with the official MicroPython setup, we've 'ported' the Getting Started with MicroPython on Pico book () examples to CircuitPython. The book is awesome, please download/purchase it to support Raspberry Pi Press ()!

Another common use of microcontrollers is in alarm systems, including home security systems. While they can get complex with a significant number of sensors, a basic motion sensing alarm is quite easy to build with a microcontroller and a few components.

This section will show you how to use your Raspberry Pi Pico board to build a basic motion sensor, and then turn it into a burglar alarm by adding lights and sound.

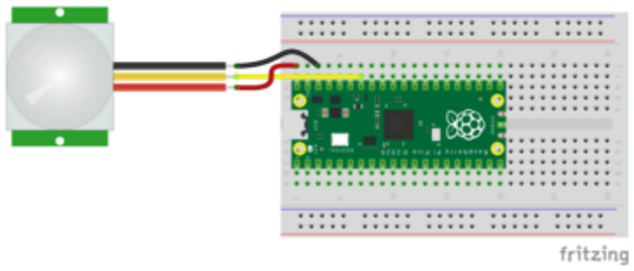
Wiring the Basic Motion Sensor

For this example you'll need your Pico board, and a PIR sensor.



PIR (motion) sensor
PIR sensors are used to detect motion from pets/humanoids from about 20 feet away (possibly works on zombies, not guaranteed). This one has an adjustable delay before firing (approx...
https://www.adafruit.com/product/189

The first step is to connect the PIR sensor to your board. Wire them up as shown below. If your PIR sensor comes with a cable, you can connect the cable and push the ends of the wires directly into the breadboard. If your PIR sensor does not have a cable, use male-to-female jumper wires to connect from the header on the sensor to the breadboard.



Board GND to PIR sensor GND
Board VBUS to PIR sensor 5V
Board GP28 to PIR sensor data pin

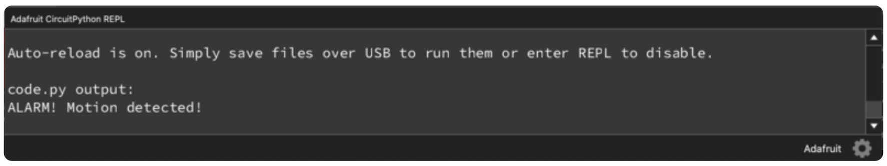
Programming the Basic Motion Sensor

Now that you've wired up your motion sensor, you can begin programming it.

Update your code.py to the following and save.

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

"""
Simple motion sensor example for Pico. Prints to serial console when PIR sensor is
triggered.

REQUIRED HARDWARE:
* PIR sensor on pin GP28.
"""
import board
import digitalio

pir = digitalio.DigitalInOut(board.GP28)
pir.direction = digitalio.Direction.INPUT

while True:
    if pir.value:
        print("ALARM! Motion detected!")
        while pir.value:
            pass
```

Now open the serial console and wave your hand in front of the sensor. Motion detected!



```
Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.

code.py output:
ALARM! Motion detected!
```

If you sit still and wait a bit before waving your hand in front of the sensor, you'll see `ALARM! Motion detected!` printed to the serial console again. However, you'll notice that if you keep waving your hand in front of the sensor, it takes a while for it to print another message. There is no delay in the code, so what's causing this behavior? The PIR sensor has a delay built in. The sensor sends a signal to your Pico board's GPIO pin when motion is detected, and maintains that signal for a period of time before stopping it. Until the signal stops, it cannot detect further motion, and therefore cannot trigger your motion detection code. Many PIR sensors have a potentiometer on them you can use to adjust the length of this delay, though there is always a minimum delay. Here is a detailed explanation ().

Now, a more detailed look at the code. First import the two necessary modules.

```
import board
import digitalio
```

Next, you set up the PIR sensor using `digitalio`. You create the object, provide it a pin, and set the pin direction to input.

```
pir = digitalio.DigitalInOut(board.GP28)
pir.direction = digitalio.Direction.INPUT
```

`pir.value` returns `True` when there is motion detected, and `False` when no motion is detected. Inside your loop, you check whether `pir.value` is `True`. If it is, you print `ALARM! Motion detected!`. Then there is a second loop checking `pir.value`, which runs continuously as long as the condition is `True`, which is a way to say "wait until `pir.value` is `False`" and then move on. When you have a loop inside of another loop, they are referred to as nested loops.

```
while True:
    if pir.value:
        print("ALARM! Motion detected!")
        while pir.value:
            pass
```

This code sequence helps the code only react to the initial change in signal from the PIR sensor, instead of reporting motion detected for the entire time the signal is sent. It means `ALARM! Motion detected!` is only printed once instead of being spammed to the serial console for the entire duration of the signal event.

Printing to the serial console is enough to test that your PIR sensor is working, but it doesn't amount to a useful alarm. Actual burglar alarms have lights and sirens that notify everyone within range that something is wrong. With a few more components, you can add the same functionality to your setup.

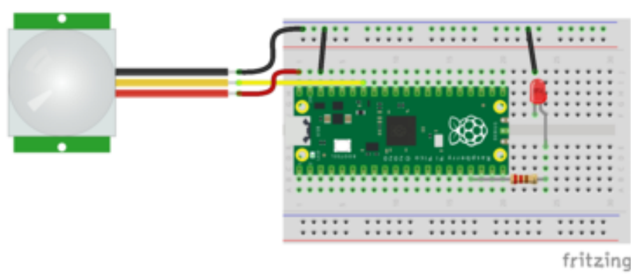
Burglar Alarm

Alarms are only useful if they have some way to notify you that they have been triggered. A blinking LED is a simple way to see when the motion sensor has been triggered. So, the first thing you'll do is add an LED to your motion sensing setup.

Wiring the Burglar Alarm with Light

This example builds on the previous. For this example, you'll need your current wiring setup, an LED of any color, a resistor, and a number of male-to-male jumper wires. A 220Ω-1.0KΩ resistor will work; a 220Ω resistor is shown in the diagram.

Connect the LED to your current setup as shown below. You'll need to modify your current setup slightly by moving the PIR sensor GND. PIR signal and power do not need to be moved.



Board GND to breadboard ground rail (using a jumper wire)
Board GP13 to 220**Ω** resistor
LED+ to 220**Ω** resistor
LED- to breadboard ground rail (using a jumper wire)
PIR sensor GND to breadboard ground rail (using a jumper wire)

Programming the Burglar Alarm with Light

Now that you've wired up your burglar alarm with light, it's time to program it.

Update your code.py to the following and save.

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

"""
A burglar alarm example for Pico. Quick flashing LED indicates alarm has been
triggered.

REQUIRED HARDWARE:
* PIR sensor on pin GP28.
* LED on pin GP13.
```

```
"""
import time
import board
import digitalio

pir = digitalio.DigitalInOut(board.GP28)
pir.direction = digitalio.Direction.INPUT
led = digitalio.DigitalInOut(board.GP13)
led.direction = digitalio.Direction.OUTPUT

motion_detected = False
while True:
    if pir.value and not motion_detected:
        print("ALARM! Motion detected!")
        motion_detected = True

    if pir.value:
        led.value = True
        time.sleep(0.1)
        led.value = False
        time.sleep(0.1)

    motion_detected = pir.value
```

Now when you save your hand in front of the sensor, as well as printing the message to the serial console, the LED will blink quickly to let you know motion has been detected! Once the sensor signal stops, the LED will stop blinking.

Now, a more detailed look at the code. First you import the same modules as before, but now you include `time` as well.

```
import time
import board
import digitalio
```

Then, in addition to the PIR sensor setup, you set up the LED.

```
pir = digitalio.DigitalInOut(board.GP28)
pir.direction = digitalio.Direction.INPUT
led = digitalio.DigitalInOut(board.GP13)
led.direction = digitalio.Direction.OUTPUT
```

You create a variable called `motion_detected` and set it to `False` before the loop.

```
motion_detected = False
```

Inside the loop, you first check to see if `pir.value` is `True` AND `motion_detected` is `False`. If BOTH of these conditions are valid, you print `ALARM! Motion detected!` to the serial console and set `motion_detected` to `True`. Next, you check if `pir.value` is `True`, and if so, you blink the LED on and off every 0.1 seconds. Finally, you set `motion_detected` equal to `pir.value`. This resets `motion_detected` to `False` once motion is no longer detected, and allows for motion to be detected again.

```
while True:
    if pir.value and not motion_detected:
        print("ALARM! Motion detected!")
        motion_detected = True

    if pir.value:
        led.value = True
        time.sleep(0.1)
        led.value = False
        time.sleep(0.1)

    motion_detected = pir.value
```

To make your alarm even more effective, you could warn intruders that the alarm is active by making the LED flash slowly when there is no motion detected. To do this, simply replace the last line of the example with an `else` block that includes code similar to the `if pir.value:` block, except with a longer `time.sleep()`.

```
[...]
    else:
        motion_detected = False
        led.value = True
        time.sleep(0.5)
        led.value = False
        time.sleep(0.5)
```

With the above update, your code.py file should look like the following.

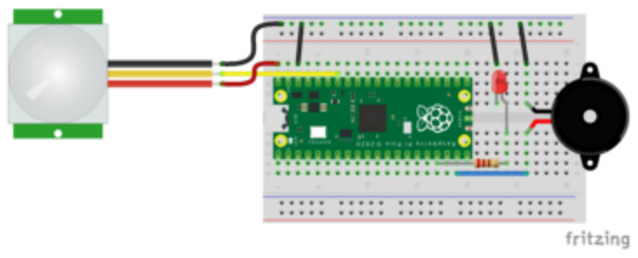
```
# SPDX-FileCopyrightText: 2021 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""
A burglar alarm example for Pico. Slow flashing LED indicates alarm is ready. Quick
flashing LED
indicates alarm has been triggered.

REQUIRED HARDWARE:
* PIR sensor on pin GP28.
* LED on pin GP13.
"""
import time
import board
import digitalio

pir = digitalio.DigitalInOut(board.GP28)
pir.direction = digitalio.Direction.INPUT
led = digitalio.DigitalInOut(board.GP13)
led.direction = digitalio.Direction.OUTPUT

motion_detected = False
while True:
    if pir.value and not motion_detected:
        print("ALARM! Motion detected!")
        motion_detected = True

    if pir.value:
        led.value = True
        time.sleep(0.1)
        led.value = False
        time.sleep(0.1)
    else:
        motion_detected = False
        led.value = True
        time.sleep(0.5)
        led.value = False
        time.sleep(0.5)
```

Your alarm system notifies everyone visually that it's running and lets you know when motion is detected. Now it needs sound!

Wiring the Burglar Alarm with Light and Sound

This example builds on the previous. For this example, you'll need your current wiring setup, and a piezo buzzer.

Connect the piezo buzzer to your current setup as shown below. Direction for the piezo buzzer does not matter.



Board GP14 to one leg of piezo buzzer
Other leg of piezo buzzer to breadboard ground rail

Programming the Burglar Alarm with Light and Sound

Now that you've wired up your burglar alarm with light and sound, it's time to program it.

Update your code.py to the following and save.

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

"""
A burglar alarm example for Pico. Slow flashing LED indicates alarm is ready. Quick
flashing LED
and beeping buzzer indicate alarm has been triggered.

REQUIRED HARDWARE:
* PIR sensor on pin GP28.
* LED on pin GP13.
* Piezo buzzer on pin GP14.
"""
import time
import board
import digitalio
import pwmio

pir = digitalio.DigitalInOut(board.GP28)
pir.direction = digitalio.Direction.INPUT
led = digitalio.DigitalInOut(board.GP13)
led.direction = digitalio.Direction.OUTPUT
buzzer = pwmio.PWMOut(board.GP14, frequency=660, duty_cycle=0,
variable_frequency=True)

motion_detected = False
while True:
    if pir.value and not motion_detected:
        print("ALARM! Motion detected!")
        motion_detected = True

    if pir.value:
        led.value = True
        buzzer.duty_cycle = 2 ** 15
        time.sleep(0.1)
        led.value = False
        buzzer.duty_cycle = 0
        time.sleep(0.1)

    else:
        motion_detected = False
        led.value = True
        time.sleep(0.5)
        led.value = False
        time.sleep(0.5)
```

Now wave your hand in front of the sensor. Motion detected, the LED blinks rapidly, and now it beeps along with the blinking for an auditory indicator that it's been triggered!

This code will look very similar to the previous example with a few modifications. First you import the necessary modules, this time including `pwmio`.

```
import time
import board
import digitalio
import pwmio
```

Next, in addition to the PIR sensor and LED setup, you set up the piezo buzzer.

```
pir = digitalio.DigitalInOut(board.GP28)
pir.direction = digitalio.Direction.INPUT
led = digitalio.DigitalInOut(board.GP13)
led.direction = digitalio.Direction.OUTPUT
buzzer = pwmio.PWMOut(board.GP14, frequency=660, duty_cycle=0,
variable_frequency=True)
```

Finally, you add two lines of code into the if block of the loop to turn the buzzer on and off along with the LED.

```
while True:
    if pir.value and not motion_detected:
        print("ALARM! Motion detected!")
        motion_detected = True

    if pir.value:
        led.value = True
        buzzer.duty_cycle = 2 ** 15
        time.sleep(0.1)
        led.value = False
        buzzer.duty_cycle = 0
        time.sleep(0.1)
```

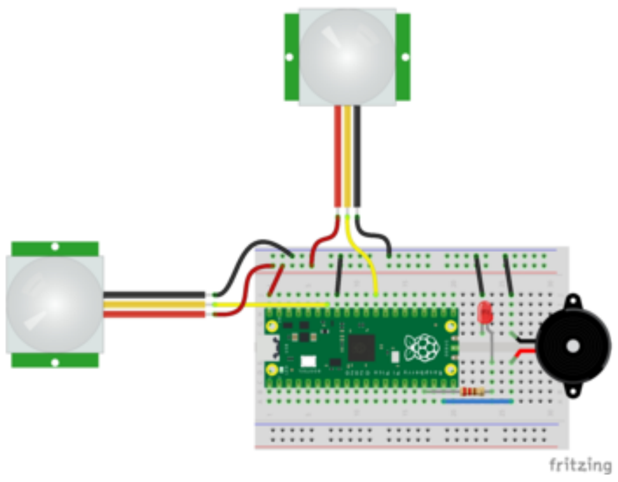
The rest of the loop remains the same. That's all there is to creating a burglar alarm with light and sound!

Home security systems rarely cover only one room or area. They are often made up of a network of many sensors connected to a single alarm system. You can easily add more sensors to your setup to monitor multiple areas at the same time.

Wiring the Extended Burglar Alarm with Light and Sound

For this example, you'll need your current wiring setup and a second PIR sensor.

Connect the second PIR sensor as shown below. Since the VBUS connection is already being used by the first PIR sensor, you'll need to make a slight modification. Move both the Pico board's VBUS connection to the breadboard power rail, and the 5V pin on the first PIR sensor to the breadboard power rail.



Board VBUS to breadboard power rail
First PIR sensor 5V to breadboard power rail
Second PIR sensor GND to breadboard ground rail
Second PIR sensor 5V to breadboard power rail
Board GP22 to second PIR sensor data pin

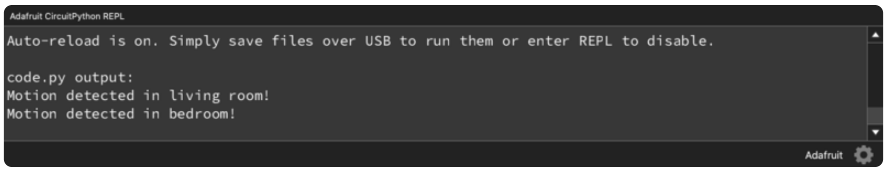
Programming the Extended Burglar Alarm

Now that you've added another sensor to your setup, it's time to program it.

Update your code.py to the following, and save.

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

"""
A burglar alarm with two motion sensors example for Pico. Slow flashing LED
indicates alarm is
ready. Quick flashing LED and beeping buzzer indicate alarm has been triggered.

REQUIRED HARDWARE:
* PIR sensor on pin GP28.
* PIR sensor on pin GP22.
* LED on pin GP13.
* Piezo buzzer on pin GP14.
"""
import time
import board
import digitalio
import pwmio

pir_one = digitalio.DigitalInOut(board.GP28)
pir_one.direction = digitalio.Direction.INPUT
pir_two = digitalio.DigitalInOut(board.GP22)
pir_two.direction = digitalio.Direction.INPUT
led = digitalio.DigitalInOut(board.GP13)
led.direction = digitalio.Direction.OUTPUT
buzzer = pwmio.PWMOut(board.GP14, frequency=660, duty_cycle=0,
variable_frequency=True)

motion_detected_one = False
motion_detected_two = False
while True:
    if pir_one.value and not motion_detected_one:
        print("ALARM! Motion detected in bedroom!")
        motion_detected_one = True

    if pir_two.value and not motion_detected_two:
        print("ALARM! Motion detected in living room!")
        motion_detected_two = True

    if pir_one.value or pir_two.value:
        led.value = True
        buzzer.duty_cycle = 2 ** 15
        time.sleep(0.1)
        led.value = False
        buzzer.duty_cycle = 0
        time.sleep(0.1)

    else:
        motion_detected_one = False
        motion_detected_two = False

        led.value = True
        time.sleep(0.5)
        led.value = False
        time.sleep(0.5)
```

Wave your hand over the first sensor to see motion reported in the bedroom. Wave your hand over the second sensor to see motion reported in the living room. Multi-area coverage!



```
Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.

code.py output:
Motion detected in living room!
Motion detected in bedroom!
```

Now a look at the code. The code will look very similar, but a number of modifications are needed to make it both work properly and be easily understood. Imports remain the same.

Setup now includes two PIR sensors, so it makes sense to update the variable names to indicate the presence of two sensors. Below the first PIR sensor, include another two lines of setup for the second one, with the appropriate pins. LED and buzzer setup remain the same.

```
pir_one = digitalio.DigitalInOut(board.GP28)
pir_one.direction = digitalio.Direction.INPUT
pir_two = digitalio.DigitalInOut(board.GP22)
pir_two.direction = digitalio.Direction.INPUT
```

Before the loop, you create two variables to track whether motion has been detected, and set them to `False` initially.

```
motion_detected_one = False
motion_detected_two = False
```

The loop begins with two if blocks. They each check a sensor value and the respective motion detected variable. If the sensor value is returning `True` and the `motion_detected` variable is `False`, then it prints the appropriate `ALARM!` message to the serial console and sets the respective `motion_detected` variable to `True`.

```
while True:
    if pir_one.value and not motion_detected_one:
        print("ALARM! Motion detected in bedroom!")
        motion_detected_one = True

    if pir_two.value and not motion_detected_two:
        print("ALARM! +Motion detected in living room!")
        motion_detected_two = True
```

Then, the code checks whether the sensors are returning True, and if they are blink the LED and beep the buzzer on and off every 0.1 seconds.

```
[...]
    if pir_one.value or pir_two.value:
        led.value = True
        buzzer.duty_cycle = 2 ** 15
        time.sleep(0.1)
        led.value = False
        buzzer.duty_cycle = 0
        time.sleep(0.1)
```

Finally, once motion is no longer detected, the motion_detected variables are set to False, and the LED blinks on and off more slowly, every 0,.5 seconds.

```
[...]
    else:
        motion_detected_one = False
        motion_detected_two = False

        led.value = True
        time.sleep(0.5)
        led.value = False
        time.sleep(0.5)
```

Now you know how to extend your alarm system to cover more than one area. You can add more sensors as needed!

Potentiometer and PWM LED

To get you started with how to program your Pico in CircuitPython, especially for those who may have started out with the official MicroPython setup, we've 'ported' the Getting Started with MicroPython on Pico book () examples to CircuitPython. The book is awesome, please download/purchase it to support Raspberry Pi Press ()!

The Pico's RP2040 microcontroller chip is a digital device, meaning by itself, it only understands digital signals, which are either on or off. Digital devices cannot understand analog(ue) signals, which can range anywhere between on and off. Therefore, included in the RP2040 is an analog(ue)-to-digital converter, or ADC. An ADC takes analog(ue) signals and converts them to digital signals. The ADC on the Pico is available on three pins: GP26, GP27, and GP28. To read analog(ue) signals, you must use one of these three pins.

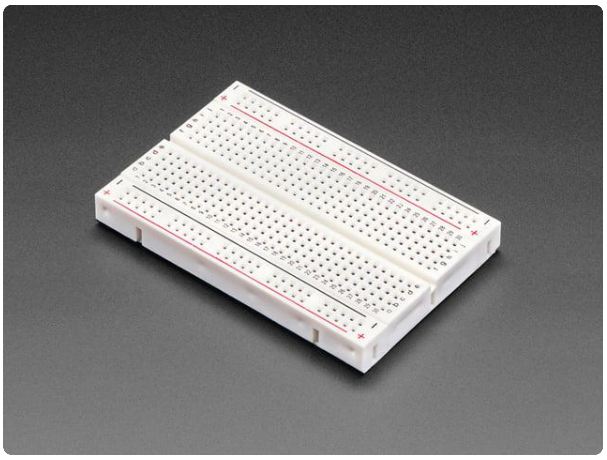
Each of the three pins can be used as an analog(ue) input, but to do that, you need an analog(ue) signal, which you can easily get from a potentiometer. There are many types of potentiometers available, such as the ones on the PIR sensor from the Burglar Alarm () section of this guide. The type of potentiometer you'll use for this example is called a "rotary potentiometer".

Reading a Potentiometer

It's simple to use CircuitPython to read the analog(ue) value from a potentiometer. The built-in module analogio does all the heavy lifting for you, and provides you with an easy way to read analog(ue) signals on analog(ue)-capable pins.

Wiring the Potentiometer

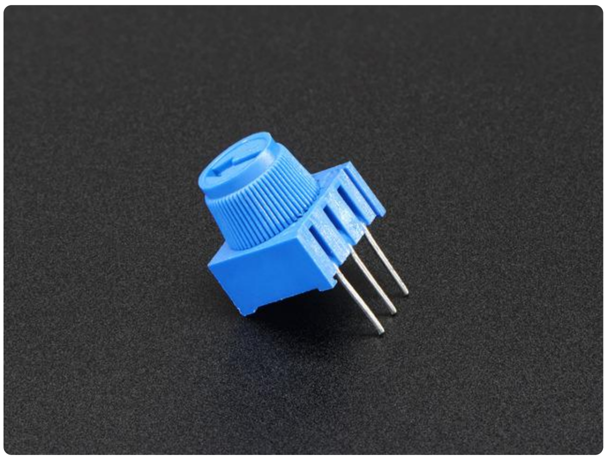
The first step is wiring your potentiometer. Connect it to your Pico as shown below. Place the potentiometer on the breadboard with the pins-side towards you to determine the proper left and right pins.



Half Sized Premium Breadboard - 400 Tie Points

This is a cute, half-size breadboard with 400 tie points, good for small projects. It's 3.25" x 2.2" / 8.3cm x 5.5cm with a standard double-strip in the...

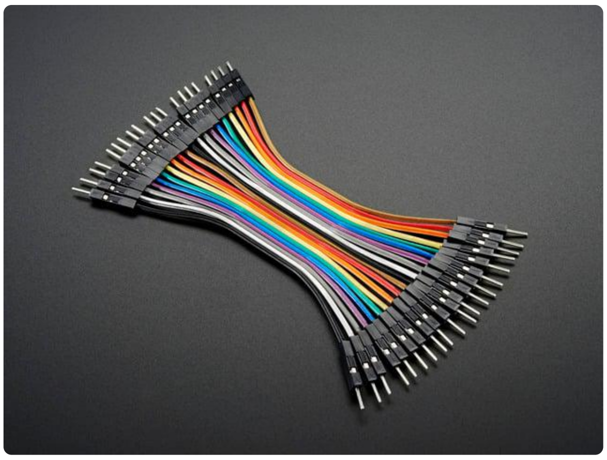
https://www.adafruit.com/product/64



Breadboard trim potentiometer

These are our favorite trim pots, perfect for breadboarding and prototyping. They have a long grippy adjustment knob and with 0.1" spacing, they plug into breadboards or...

https://www.adafruit.com/product/356

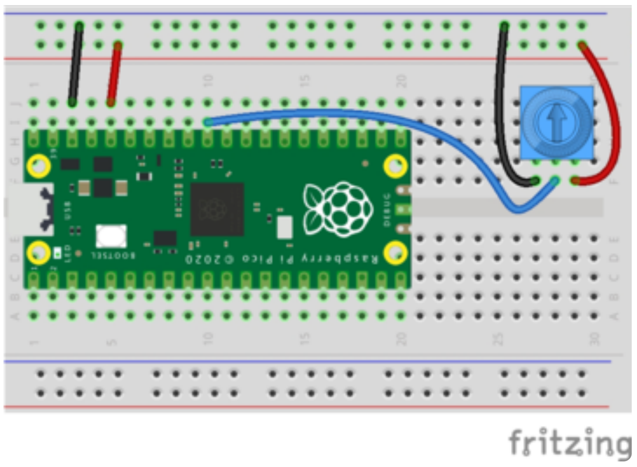


Premium Male/Male Jumper Wires - 20 x 3" (75mm)

Handy for making wire harnesses or jumpering between headers on PCB's. These premium jumper wires are 3" (75mm) long and come in a 'strip' of 20 (2 pieces of each...

https://www.adafruit.com/product/1956

For this example, you'll need your Pico, a potentiometer, and some male-to-male jumper wires.



Board GND to breadboard ground rail
Board 3V3 to breadboard power rail
Board GP26 to potentiometer middle leg
Potentiometer left leg to breadboard ground rail
Potentiometer right leg to breadboard power rail

Programming to Read the Potentiometer

Update your code.py to the following and save.

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

"""
Read the potentiometer value. Prints the value to the serial console every two
seconds.

REQUIRED HARDWARE:
* potentiometer on pin GP26.
"""
import time
import board
import analogio

potentiometer = analogio.AnalogIn(board.GP26)

while True:
    print(potentiometer.value)
    time.sleep(2)
```

Now, connect to the serial console. You should see a value between 0 and 65535, depending on the current rotation of the knob, being printed out to the serial console. Try turning the knob on the potentiometer. When you turn it to the left, the value goes down, and when you turn it to the right, the value goes up.

If the values go the opposite direction when you turn the knob, try swapping the ground and power connections to your potentiometer. You may have them backwards!

Don't worry if your value never quite reaches 0 or 65535. Electronic components are built with what's called a tolerance, which means the resulting values may not be precise.

This value is the analog value. While you can say to yourself that 65535 is max voltage and 0 is min voltage, without doing some math in your head, the rest of the values aren't that useful to you. It's much easier to let CircuitPython do the math for you.

Update your code.py to the following and save.

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

"""
Convert the potentiometer value to a voltage value. Prints the voltage value to the
serial console
every two seconds.

REQUIRED HARDWARE:
* potentiometer on pin GP26.
"""
import time
import board
import analogio

potentiometer = analogio.AnalogIn(board.GP26)

get_voltage = 3.3 / 65535

while True:
    voltage = potentiometer.value * get_voltage
    print(voltage)
    time.sleep(2)
```

The `get_voltage` equation provides a reasonable approximation of the voltage it represents. The first number is the maximum voltage available from the 3V3 pin on your Pico, and the second number is the maximum analog value. Essentially, dividing 3.3 / 65535 returns a voltage value based on the analog value.

To use the equation, you simply multiply the raw potentiometer value by the `get_voltage` helper, and print the resulting value to the serial console. Now if you check the serial console, you'll see a number between 0 and 3.3, depending on the current rotation of the knob. Turn the knob to see the values change!

That's all there is to reading the values of a potentiometer as both a raw analog(ue) value and a voltage value using CircuitPython and Pico!

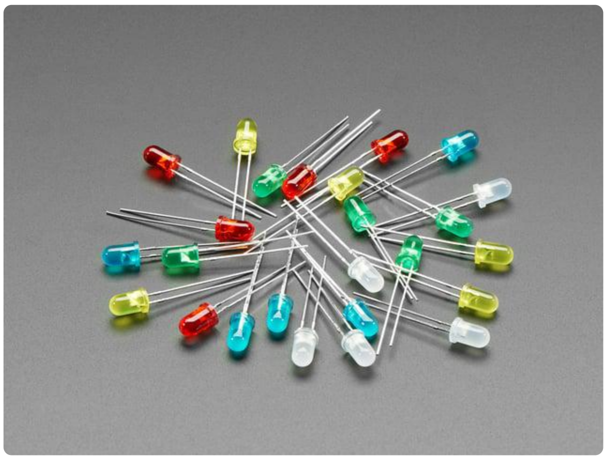
Using PWM to Fade an LED

So far, everything you've done with LEDs have involved an on or off state. This is because the digital outputs on a microcontroller and only be on or off. Turning these outputs on and off is known as a pulse, and depending on the speed at which you change the pin state, you can "modulate" these pulses using Pulse Width Modulation. PWM has many uses. This example will show you how to use PWM to fade an LED up and down.

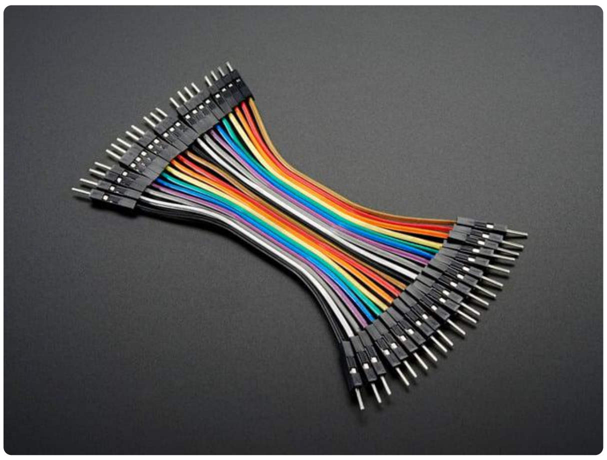
Every pin on the Pico can do PWM, however, you cannot do PWM on every pin at the same time. Some sets of pins on the Pico use the same PWM output, meaning they cannot be used a the same time to create a PWM object. If you want to know which pins use the same outputs, you can read the datasheet, or you can try to create a PWM object using CircuitPython. If you try to create a PWM object on two conflicting pins, CircuitPython will give you a `ValueError: All timers for this pin are in use` error. If this happens, choose a different pin.

Wiring the LED

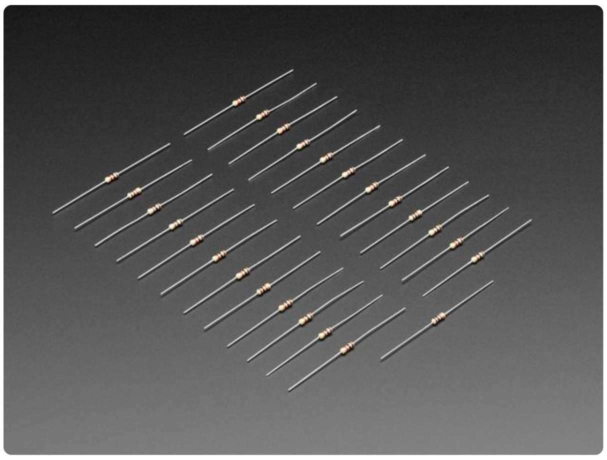
The first step is to add an LED to your existing potentiometer setup.



Diffused 5mm LED Pack - 5 LEDs each in 5 Colors - 25 Pack
Need some indicators? We are big fans of these diffused LEDs. They are fairly bright, so they can be seen in daytime, and from any angle. They go easily into a breadboard and will add...
https://www.adafruit.com/product/4203



Premium Male/Male Jumper Wires - 20 x 3" (75mm)
Handy for making wire harnesses or jumpering between headers on PCB's. These premium jumper wires are 3" (75mm) long and come in a 'strip' of 20 (2 pieces of each...
https://www.adafruit.com/product/1956

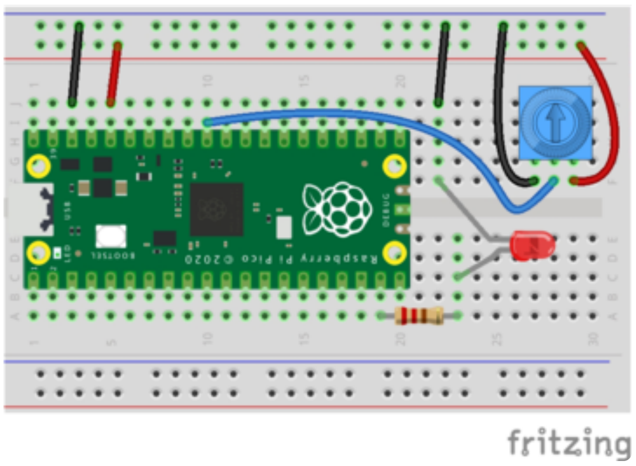


Through-Hole Resistors - 1.0K ohm 5% 1/4W - Pack of 25

ΩMG! You're not going to be able to resist these handy resistor packs! Well, axially, they do all of the resisting for you!This is a 25 Pack of...

https://www.adafruit.com/product/4294

For this example, you'll need your existing wiring setup, an LED, a resistor, and a male-to-male jumper wire. A 220Ω-1.0KΩ resistor will work; a 220Ω resistor is shown in the diagram.



Board GP14 to 220**Ω** resistor
LED+ to 220**Ω** resistor
LED- to breadboard ground rail (using a jumper wire)

Programming to Fade the LED

Update your code.py to the following and save.

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

"""
Use PWM to fade an LED up and down using the potentiometer value as the duty cycle.

REQUIRED HARDWARE:
* potentiometer on pin GP26.
* LED on pin GP14.
"""
import board
import analogio
import pwmio

potentiometer = analogio.AnalogIn(board.GP26)
led = pwmio.PWMOut(board.GP14, frequency=1000)
```

```
while True:
    led.duty_cycle = potentiometer.value
```

Now try turning the potentiometer. As you turn it, the LED will fade up or down depending on the direction you're rotating the knob. When the knob is all the way to the left, the LED will be off. When it is all the way to the right, it will be brightest.

When creating the PWM object, you provide both the pin you connected the LED to and a frequency. Then, inside the loop, we set the duty cycle equal to the potentiometer raw analog(ue) value.

Duty cycle controls the pin's output. At 0% duty cycle, the pin is off. At 100% duty cycle, the pin is fully on. A duty cycle of 50% means that the pin is on for half the pulses, and off for half. The reading taken from the potentiometer is applied to the LED PWM duty cycle, so a low reading is like a low voltage on an analog(ue) input which means the LED will be dim, and a high reading is like a high voltage which means the LED will be bright.

That's all there is to using PWM with CircuitPython and Pico to fade an LED!

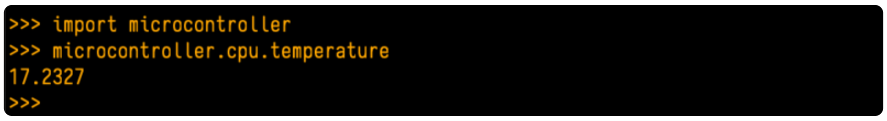
Temperature Gauge

To get you started with how to program your Pico in CircuitPython, especially for those who may have started out with the official MicroPython setup, we've 'ported' the Getting Started with MicroPython on Pico book () examples to CircuitPython. The book is awesome, please download/purchase it to support Raspberry Pi Press ()!

The Raspberry Pi Pico's CPU has a temperature sensor built into it. CircuitPython makes it super simple to read this data from the sensor using the `microcontroller` module.

Plug in your Pico board, open Mu, click into the serial console, and press CTRL+C followed by ENTER to enter the REPL. At the REPL prompt (`>>>`), type the following:

```
import microcontroller
microcontroller.cpu.temperature
```



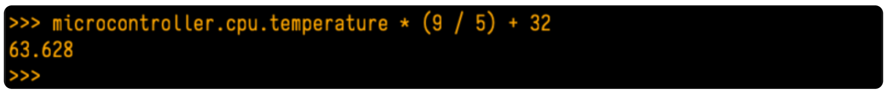
```
>>> import microcontroller
>>> microcontroller.cpu.temperature
17.2327
>>>
```

This returns the temperature of the sensor in Celsius. Note that it's not exactly the ambient temperature and it's not super precise. But it's close!

Try holding your finger over the CPU, the large black square in the middle of your Pico board. Then, check the temperature again. The returned temperature has likely increased because you warmed the chip slightly!

If you'd like to print it out in Fahrenheit, use this simple formula: Fahrenheit = Celsius * (9/5) + 32. It's super easy to do math using CircuitPython. Check it out!

```
microcontroller.cpu.temperature * (9 / 5) + 32
```



```
>>> microcontroller.cpu.temperature * (9 / 5) + 32
63.628
>>>
```

That's all there is to reading the temperature data from the temperature sensor built into the CPU on your Raspberry Pi Pico using CircuitPython!

Data Logger

To get you started with how to program your Pico in CircuitPython, especially for those who may have started out with the official MicroPython setup, we've 'ported' the Getting Started with MicroPython on Pico book () examples to CircuitPython. The book is awesome, please download/purchase it to support Raspberry Pi Press ()!

All of the projects in this guide have used your Raspberry Pi Pico while connected to your computer. However, there are many microcontroller projects that work standalone, powered from the wall or a battery pack, and your Pico is perfectly capable of doing the same.

This section will show you how to use CircuitPython to read the internal temperature data and write it to a file on the filesystem to create a temperature data logger.

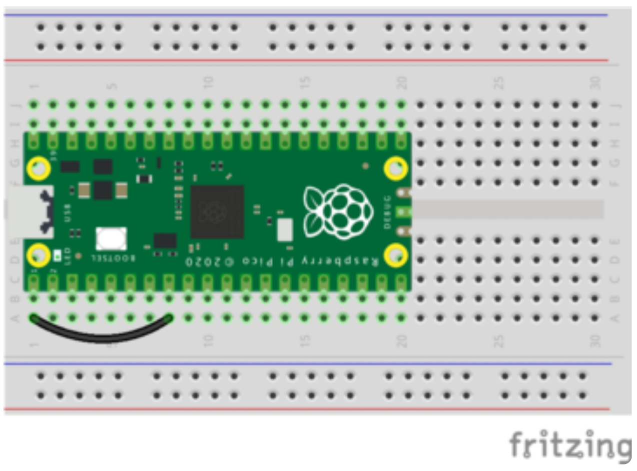
For this example, you'll need your Pico board, and if you want to use it separated from your computer, a micro USB wall charger or a USB battery pack and a micro USB cable.

Data Logger Wiring

CircuitPython does not allow your computer to write to the filesystem at the same time as CircuitPython is writing to the filesystem. Therefore, if you simply run `storage.remount("/", readonly=False)` in boot.py, CIRCUITPY is no longer writable by your computer, which means you cannot write to or delete files from the C IRCUITPY drive. This means you cannot modify boot.py. To return the filesystem to a state writable by your computer, you would need to run some commands in the REPL. More information on this process is available at the end of this page.

Alternatively, you can add a little extra code to boot.py that only runs `remount` if a particular pin is connected to a ground pin. This means if you start up the board with the specified pin connected to ground, the filesystem will be read-only to your computer (and therefore writable by CircuitPython), and if you start up the board without the pin connected to ground, the filesystem will be writable by your computer. If you include this extra code, you will need to have a jumper wire handy. The boot.py shown below contains this extra code. So, when you want to begin data logging, you'll need to connect a jumper wire as shown below before plugging the Pico into USB.

For this example, you'll need your Pico, a breadboard, and a male-to-male jumper wire.



Pico GP0 to Pico GND

Programming the Temperature Data Logger

Start with the jumper wire disconnected.

First create a file called boot.py on your CIRCUITPY drive, and save it with the following contents.

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

"""
boot.py file for Pico data logging example. If pin GP0 is connected to GND when
the pico starts up, make the filesystem writeable by CircuitPython.
"""
import board
import digitalio
import storage

write_pin = digitalio.DigitalInOut(board.GP0)
write_pin.direction = digitalio.Direction.INPUT
write_pin.pull = digitalio.Pull.UP

# If write pin is connected to ground on start-up, CircuitPython can write to
CIRCUITPY filesystem.
if not write_pin.value:
    storage.remount("/", readonly=False)
```

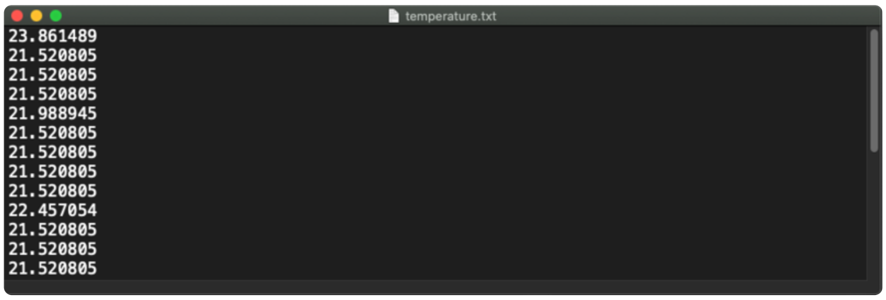
It is important to know that the boot.py file is run only when the Pico starts up, e.g. when it is plugged into USB or power. (It is NOT run when the board soft resets, e.g, when you save a file etc.) This code sets up pin GP0 as an input with a pull-up.

Update your code.py to the following and save.

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

"""
Data logging example for Pico. Logs the temperature to a file on the Pico.
"""
import time
import board
import digitalio
import microcontroller

led = digitalio.DigitalInOut(board.LED)
led.switch_to_output()

try:
    with open("/temperature.txt", "a") as datalog:
        while True:
            temp = microcontroller.cpu.temperature
            datalog.write('{0:f}\n'.format(temp))
            datalog.flush()
            led.value = not led.value
            time.sleep(1)
except OSError as e:  # Typically when the filesystem isn't writeable...
    delay = 0.5  # ...blink the LED every half second.
    if e.args[0] == 28:  # If the filesystem is full...
        delay = 0.25  # ...blink the LED faster!
    while True:
        led.value = not led.value
        time.sleep(delay)
```

The LED should begin blinking every 0.5 seconds. This lets you know the code is running but nothing else is happening. Since you started up the board with the wire disconnected, no data logging is occurring.

Disconnect your Pico from USB, and plug the jumper wire in as shown above. Then plug your board into USB. The LED blinking will slow down to every 1 second. CIRCUI TPY will mount, but only as a read-only filesystem. You'll see a temperature.txt file that wasn't there before. If you view this file, you'll see the beginning of a list of temperatures. The temperature is checked every second and added to the file.


temperature.txt

```
23.861489
21.520805
21.520805
21.520805
21.988945
21.520805
21.520805
21.520805
21.520805
22.457054
21.520805
21.520805
21.520805
```

You can now let it run to track an ambient temperature in the area. If you use a USB battery pack, you can place it anywhere to track temperature.

If you let it run for long enough, you can fill up the filesystem. At that time, the LED will start blinking quickly, every 0.25 seconds, to let you know.

When you're ready to work with your Pico again, you can unplug it from USB, unplug the jumper wire, and plug the Pico into your computer's USB again. CIRCUITPY will once again be writable by your computer and you can update code as needed.

That's all there is to logging the ambient temperature using CircuitPython and Pico!

Data Logger Without Wiring

If you want to use your Pico without a breadboard for data logging, or simply don't want to connect a wire between a pin and ground, you would use the following boot.p y file.

If you add this file to CIRCUITPY, your board will always be read-only to your computer when it is plugged into USB.

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

"""
boot.py file for Pico data logging example. If this file is present when
the pico starts up, make the filesystem writeable by CircuitPython.
"""
import storage

storage.remount("/", readonly=False)
```

This will make the filesystem writable by CircuitPython for the purposes of data logging every time the board is powered up. This means you can plug your Pico into your computer's USB or a USB battery pack with no additional wiring necessary, and the temperature logging will begin.

However, you'll notice that you cannot edit any of the files on the CIRCUITPY drive. So how do you make the filesystem writable by your computer again? You can use the REPL.

Using the REPL to Rename boot.py

Even when you can't write to CIRCUITPY, you can still get to the REPL. Connect your Pico to your computer via USB, and connect to the serial console. Enter the REPL (`>>>`) and one of the following commands.

To rename boot.py to something else so your Pico no longer sees it, run the following.

```
import os
os.rename("boot.py", "boot1.py")
```

To remove boot.py entirely, run the following.

```
import os
os.remove("boot.py")
```

That's all there is to data logging with CircuitPython and Pico without needing a jumper wire!

NeoPixel LEDs

To get you started with how to program your Pico in CircuitPython, especially for those who may have started out with the official MicroPython setup, we've 'ported' the Getting Started with MicroPython on Pico book () examples to CircuitPython. The book is awesome, please download/purchase it to support Raspberry Pi Press ()!

It's easy to control addressable RGB NeoPixel LEDs with the Raspberry Pi Pico, CircuitPython and the Adafruit CircuitPython NeoPixel () library. Simply connect the LEDs to your Pico board, copy the library to your CIRCUITPY drive, and update your code.py file. That's all there is to it.

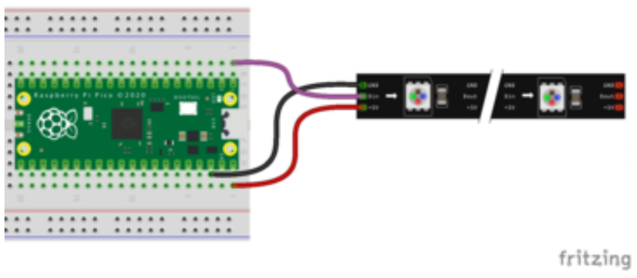
This section will show you how to wire up a NeoPixel LED strip to your Pico board, install the NeoPixel library, and walk through a few examples of controlling the LEDs.

Adafruit NeoPixel LED Strip w/ Alligator Clips - 60 LED/m
Adding glowy color to your projects has never been easier: no more soldering or stripping wires, clip 'em on and glow! This Adafruit NeoPixel LED Strip with Alligator...
https://www.adafruit.com/product/3811

Wiring the NeoPixel LED Strip

The first step is to connect the NeoPixel LEDs to your Pico board. Wire them as shown below.

Note that while your Pico board can control a considerable number of NeoPixel LEDs, the power you can draw from the 5V VBUS pin on your Pico board is limited. The examples below assume a strip of 30 NeoPixel LEDs, which the Pico board is capable of handling. However, if you want to control more than that, it is suggested that you use an external power supply to power your LEDs. Check out the NeoPixel guide () for everything there is to know about NeoPixel LEDs, including power requirements and suggestions.



Board GND to NeoPixel GND
Board VBUS to NeoPixel 5V (power)
Board GP0 to NeoPixel signal

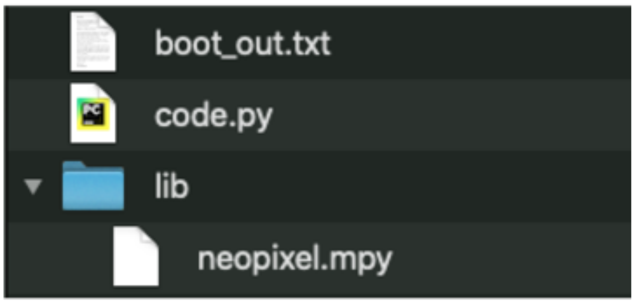
Installing the Adafruit CircuitPython NeoPixel Library

You'll need to install the Adafruit CircuitPython NeoPixel () library on your CircuitPython board.

Next you'll need to install the necessary library to use the NeoPixels -- carefully follow the steps to find and install this library from Adafruit's CircuitPython library bundle (). Our CircuitPython starter guide has a great page on how to install the library bundle ().

Copy the following file from the bundle to the lib folder on your CIRCUITPY drive:

- neopixel.mpy



Before continuing make sure your board's lib folder or root filesystem has the neopixel.mpy file copied over.

Programming NeoPixel LEDs

Now that you've wired up your NeoPixel LEDs and loaded the NeoPixel library onto your CIRCUITPY drive, it's time to begin programming.

Update your code.py to the following. You should verify that `num_pixels` matches the number of NeoPixel LEDs you have connected to your board, if you connected a different form factor or strip length to your Pico board.

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

"""
NeoPixel example for Pico. Turns the NeoPixels red.

REQUIRED HARDWARE:
* RGB NeoPixel LEDs connected to pin GP0.
"""
import board
import neopixel

# Update this to match the number of NeoPixel LEDs connected to your board.
num_pixels = 30

pixels = neopixel.NeoPixel(board.GP0, num_pixels)
pixels.brightness = 0.5
```

```
while True:
    pixels.fill((255, 0, 0))
```

The NeoPixels light up red!

Now a quick look at the code. First, you import the necessary module and library.

```
import board
import neopixel
```

Modules are built into CircuitPython. Libraries are separate files or groups of files saved to the CIRCUITPY drive. Regardless, you import them both the same way.

Next you create a variable to set the number of pixels, which defaults to `30`.

```
num_pixels = 30
```

Then, you set up the NeoPixels. The `neopixel` object requires two things: the pin the NeoPixels are connected to, and the number of NeoPixels connected. In this case, you connected the strip to GP0 on your Pico board, and you set the number of pixels in the previous line of code.

```
pixels = neopixel.NeoPixel(board.GP0, num_pixels)
```

Following that, you set the brightness of the NeoPixels to `0.5`, or 50%. `brightness` expects a float between 0.0 and 1.0, where 0 is off, and 1 is 100% brightness. Point the NeoPixels away from you, or put a sheet of paper over them to diffuse them a bit, and try setting brightness to 1. They can get really bright! That's why we set the brightness to half, which is still quite bright. You can set it even lower if you like.

```
pixels.brightness = 0.5
```

LED colors are set using a combination of red, green, and blue, often in the form of an (R, G, B) tuple. Each member of the tuple is set to a number between `0` and `255` that determines the amount of each color present. Red, green and blue in different combinations can create all the colors in the rainbow! So, for example, to set the LED to red, the tuple would be `(255, 0, 0)`, which has the maximum level of red, and no green or blue. Green would be `(0, 255, 0)`, etc. For the colors between, you set a combination, such as cyan which is `(0, 255, 255)`, with equal amounts of green and blue.

Inside your loop, you use the `fill()` function to "fill", or turn on, the NeoPixels a specified color. The `fill()` function works with both RGB tuple colors, and hex colors. This example turns them on red.

```
while True:
    pixels.fill((255, 0, 0))
```

Note the double parentheses - this is specific to setting the color using an RGB tuple. The `fill()` function expects one argument, and the entire tuple, including its parentheses, is that single argument. Without the parentheses around the tuple, `fill()` will see it as three arguments separated by commas, and your code will not run.

Now, try turning them green. Update the loop to the following:

```
while True:
    pixels.fill((0, 255, 0))
```

Green!

Now try blue.

```
while True:
    pixels.fill((0, 0, 255))
```

Easy!

What if you wanted to make them light up the three colors in sequence without manually updating your code each time? For that, you'll need `time`.

Update your code.py to the following.

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

"""
NeoPixel example for Pico. Turns the NeoPixels red, green, and blue in sequence.

REQUIRED HARDWARE:
* RGB NeoPixel LEDs connected to pin GP0.
"""
import time
import board
import neopixel

# Update this to match the number of NeoPixel LEDs connected to your board.
num_pixels = 30

pixels = neopixel.NeoPixel(board.GP0, num_pixels)
pixels.brightness = 0.5
```

```
while True:
    pixels.fill((255, 0, 0))
    time.sleep(0.5)
    pixels.fill((0, 255, 0))
    time.sleep(0.5)
    pixels.fill((0, 0, 255))
    time.sleep(0.5)
```

You can change the colors by editing the color tuples, and the timing by updating the number of seconds given to `time.sleep()`.

For example, update the loop to the following for three new colors and a slower change.

```
while True:
    pixels.fill((255, 255, 0))
    time.sleep(1)
    pixels.fill((0, 255, 255))
    time.sleep(1)
    pixels.fill((255, 0, 255))
    time.sleep(1)
```

And finally, what about rainbows? Rainbows involve using `colorwheel` and some math.

Update your code.py to the following.

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

"""
NeoPixel example for Pico. Displays a rainbow on the NeoPixels.

REQUIRED HARDWARE:
* RGB NeoPixel LEDs connected to pin GP0.
"""
import time
import board
from rainbowio import colorwheel
import neopixel

# Update this to match the number of NeoPixel LEDs connected to your board.
num_pixels = 30

pixels = neopixel.NeoPixel(board.GP0, num_pixels, auto_write=False)
pixels.brightness = 0.5


def rainbow(speed):
    for j in range(255):
        for i in range(num_pixels):
            pixel_index = (i * 256 // num_pixels) + j
            pixels[i] = colorwheel(pixel_index & 255)
        pixels.show()
        time.sleep(speed)
```

```
while True:
    rainbow(0)
```

In addition to the three previous imports, we also import `colorwheel` from the built-in `_pixelbuf` module.

```
from _pixelbuf import colorwheel
```

Then, there's a change to the pixel setup. Note, that in the `neopixel` object, you now set `auto_write=False`. This means that when you tell the pixels to do something, nothing will happen unless you call `pixels.show()`. This is necessary for the rainbow function to work. Be aware that if you add in some of the code from above, you'll have to call `pixels.show()` after filling the pixels a given color or they won't turn on!

```
pixels = neopixel.NeoPixel(board.GP0, num_pixels, auto_write=False)
```

If you write some NeoPixel code, and your pixels aren't lighting up, check that auto_write isn't set to False, or make sure you call pixels.show()!

Next comes the `rainbow()` function. This function takes one argument, `speed` in seconds.

```
def rainbow(speed):
    for j in range(255):
        for i in range(num_pixels):
            pixel_index = (i * 256 // num_pixels) + j
            pixels[i] = colorwheel(pixel_index &amp; 255)
        pixels.show()
        time.sleep(speed)
```

Finally, you call the `rainbow()` function in a loop. For fast rainbows, set it to `0`. To slow it down, increase the number.

```
while True:
    rainbow(0)
```

Rainbows!

That's all there is to basic NeoPixel LED control.

If you want to get really fancy and take it further, you can check out the guide on the Adafruit CircuitPython LED Animation library (), which shows how the library makes displaying animations on RGB LEDs super simple.

FAQ and Troubleshooting

FAQ

board.I2C(), board.SPI(), and board.UART() do not exist. What should I do?

The Pico does not have specific pins labeled on the board as the default I2C, SPI, or UART pins. See the Pinouts () page for more details.

Is pulseio supported?

pulseio is not yet supported. Please subscribe to this CircuitPython issue () to be notified when it's been added.

CircuitPython Essentials

CircuitPython Essentials ()

Downloads

- Raspberry Pi: Getting Started with your Pico ()
- RP2040 Datasheet ()
- Hardware Design with RP2040 ()
- Raspberry Pi Pico Datasheet ()
- Getting Started with Raspberry Pi Pico ()
- Pico C/C++ SDK ()
- Pico Python SDK () (MicroPython, not CircuitPython)

Here's a paper template () to cut out and use under your Pico to see the pinouts. Print this PDF out at exact size (do not use Shrink to Fit or Fit to Page options). Punch holes in the template using jumpers as needed. To punch the pins through initially, try placing the paper underneath the Pico on top of some anti-static foam.