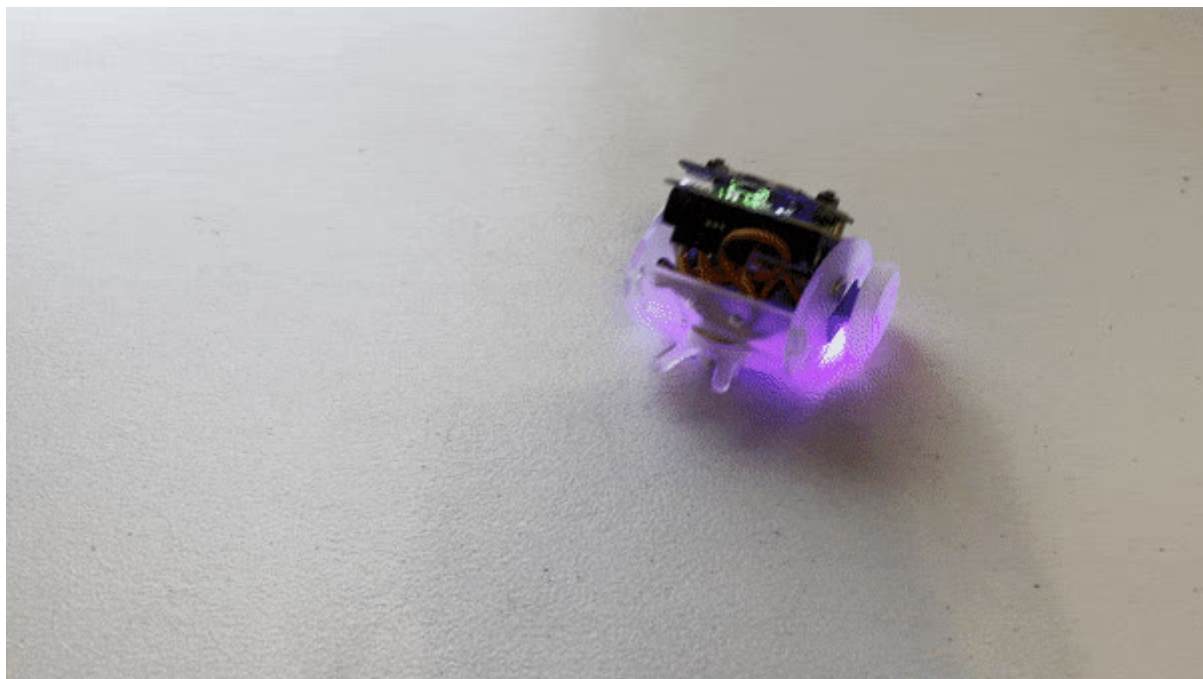




Bluetooth CLUE Robot Car using CircuitPython

Created by Melissa LeBlanc-Williams



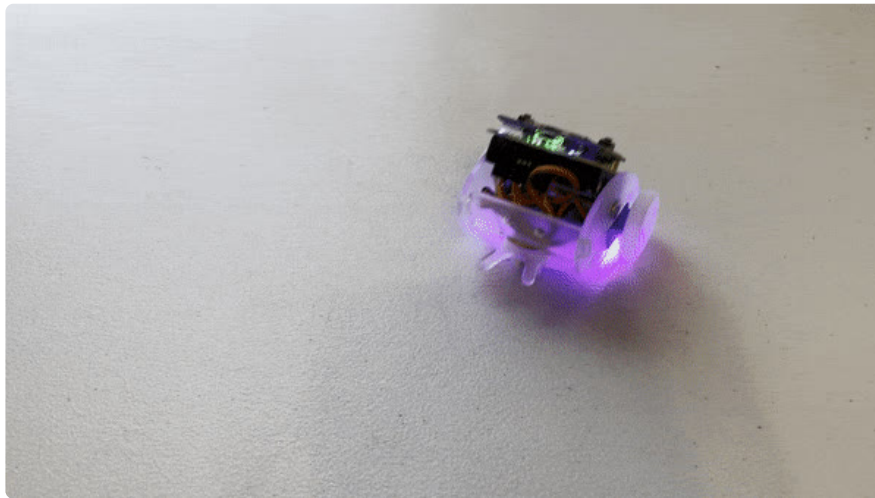
<https://learn.adafruit.com/bluetooth-clue-robot-car-using-circuitpython>

Last updated on 2025-02-24 03:31:05 PM EST

Table of Contents

Overview	3
<hr/>	
• Required Parts	
CircuitPython on CLUE	5
<hr/>	
• Set up CircuitPython Quick Start!	
Setup	7
<hr/>	
• Assemble The Car Kit	
• Install Bluefruit Connect App	
Code the Robot	10
<hr/>	
• Code and Libraries	
• Upload the Code and Libraries to the CLUE	
• Code Walkthrough	
Usage	22
<hr/>	
• Connect with Bluefruit App	
• Driving the Robot	
• Turning On and Off Lights	
• Changing Underglow Color	
• Adding More Sensors	

Overview



Using the Adafruit CLUE with the Ring:Bit Car allows you to build a robotic car with access to many sensors. When paired with CircuitPython, this makes for a friendly project that is a great way to get started with robotics.

The code allows you to connect to the robot using a Bluetooth device such as a phone or tablet running [Adafruit Bluefruit Connect \(https://adafru.it/DNc\)](https://adafru.it/DNc) in order to remotely send input. This allows you to send commands to direct it where you would like it to go, change the color of the underglow lighting, and even turn the CLUE light on and off.

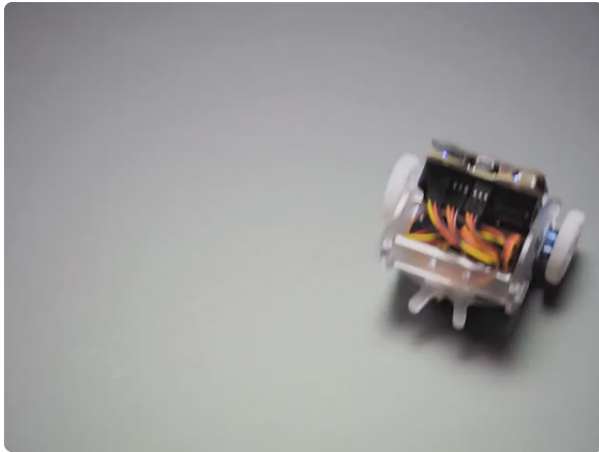
For this project, in order to keep it simple, only the display, lights, and motors were used along with the Bluetooth capabilities. This project could be further extended to make use of the [many additional sensors available \(https://adafru.it/K7b\)](https://adafru.it/K7b) on the CLUE if you desired.

For the graphics, everything was drawn in code using the CircuitPython built-in [vectorio \(https://adafru.it/ZfG\)](https://adafru.it/ZfG) module. This helps to reduce the number of dependent files needed to run the robot, increases the drawing speed, and makes it easier to modify the graphics.

This robot was featured on a CircuitPython Day livestream build.

Required Parts

This project only requires a couple of parts:



micro:bit Ring:Bit Buggy Car Robot (without micro:bit)

The micro:bit Ring:Bit Buggy Car Robot is a small and cute DIY smart car based around the BBC micro:bit. It's a...

<https://www.adafruit.com/product/4442>



Adafruit CLUE - nRF52840 Express with Bluetooth LE

Do you feel like you just don't have a CLUE? Well, we can help with that - get a CLUE here at Adafruit by picking up this sensor-packed development board. We wanted to build some...

<https://www.adafruit.com/product/4500>

This robot also requires 3 AAA-sized batteries.



Alkaline AAA batteries - 3 pack

Battery power for your portable project! These batteries are good quality at a good price, and work fantastic with any of the kits or projects in the shop that use AAA's. This is a...

<https://www.adafruit.com/product/3520>



USB cable - USB A to Micro-B

This here is your standard A to micro-B USB cable, for USB 1.1 or 2.0. Perfect for connecting a PC to your Metro, Feather, Raspberry Pi or other dev-board or...

<https://www.adafruit.com/product/592>

CircuitPython on CLUE

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

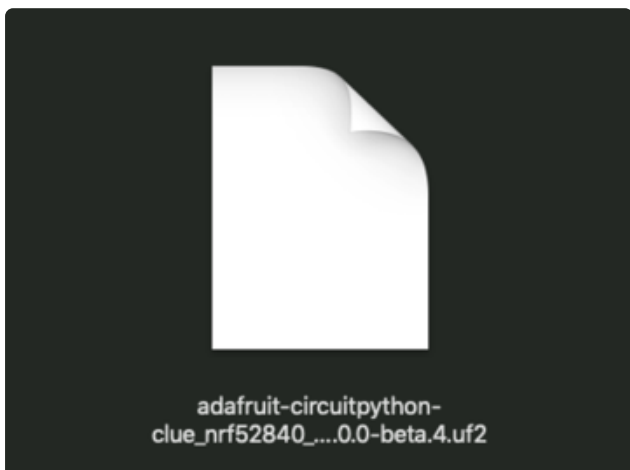
The following instructions will show you how to install CircuitPython. If you've already installed CircuitPython but are looking to update it or reinstall it, the same steps work for that as well!

Set up CircuitPython Quick Start!

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

Download the latest version of
CircuitPython for CLUE from
circuitpython.org

<https://adafru.it/IHF>



Click the link above to download the latest version of CircuitPython for the CLUE.

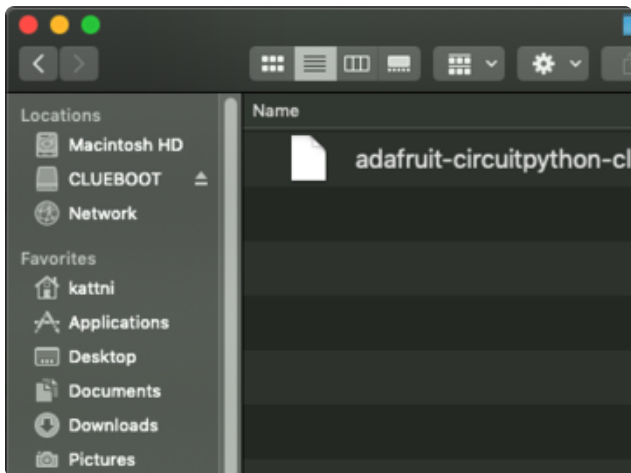
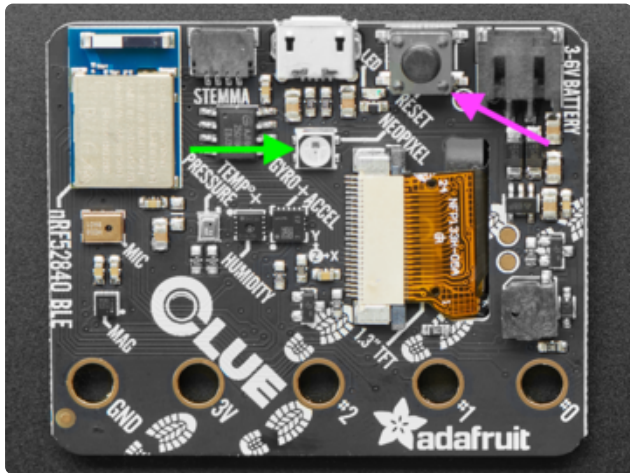
Download and save it to your desktop (or wherever is handy).

Plug your CLUE into your computer using a known-good USB cable.

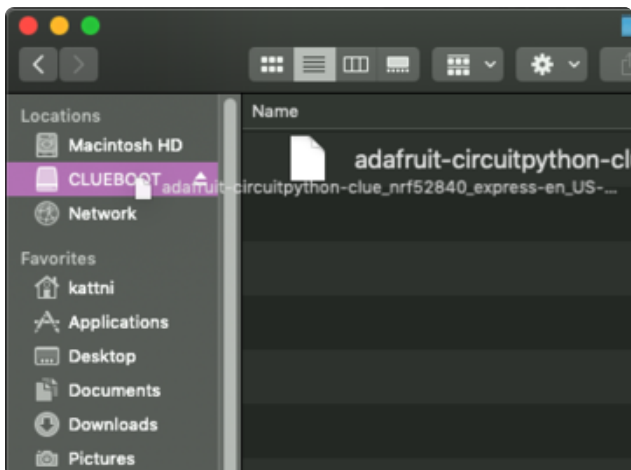
A lot of people end up using charge-only USB cables and it is very frustrating! So make sure you have a USB cable you know is good for data sync.

Double-click the **Reset** button on the top (magenta arrow) on your board, and you will see the NeoPixel RGB LED (green arrow) turn green. If it turns red, check the USB cable, try another USB port, etc. **Note:** The little red LED next to the USB connector will pulse red. That's ok!

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

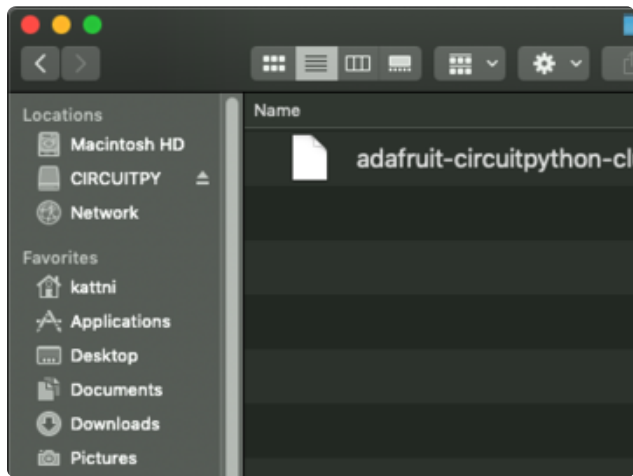


You will see a new disk drive appear called **CLUEBOOT**.



Drag the **adafruit-circuitpython-clue-etc.uf2** file to **CLUEBOOT**.

The LED will flash. Then, the **CLUEBOOT** drive will disappear and a new disk drive called **CIRCUITPY** will appear.



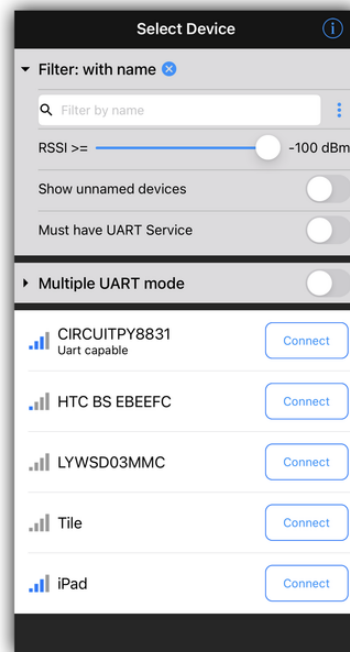
If this is the first time you're installing CircuitPython or you're doing a completely fresh install after erasing the filesystem, you will have two files - **boot_out.txt**, and **code.py**, and one folder - **lib** on your **CIRCUITPY** drive.

If CircuitPython was already installed, the files present before reloading CircuitPython should still be present on your **CIRCUITPY** drive. Loading CircuitPython will not create new files if there was already a CircuitPython filesystem present.

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

Setup

Setting up the robot is fairly simple and only requires a couple of prerequisites outside of getting the code running. The car comes in a kit form and includes the tools and instructions you need to assemble it. You will also need the [Adafruit Bluefruit Connect \(https://adafru.it/DNc\)](https://adafru.it/DNc) app running on a phone or tablet to send commands.



Assemble The Car Kit

The first thing you will need to do is assemble the Ring:bit Car V2. Adafruit previously sold a version branded as the Pi Supply bit:buggy, which is functionally the same. Each car kit includes assembly instructions, or if you would prefer, you can follow along with the [online instructions](https://adafru.it/11By) (<https://adafru.it/11By>).

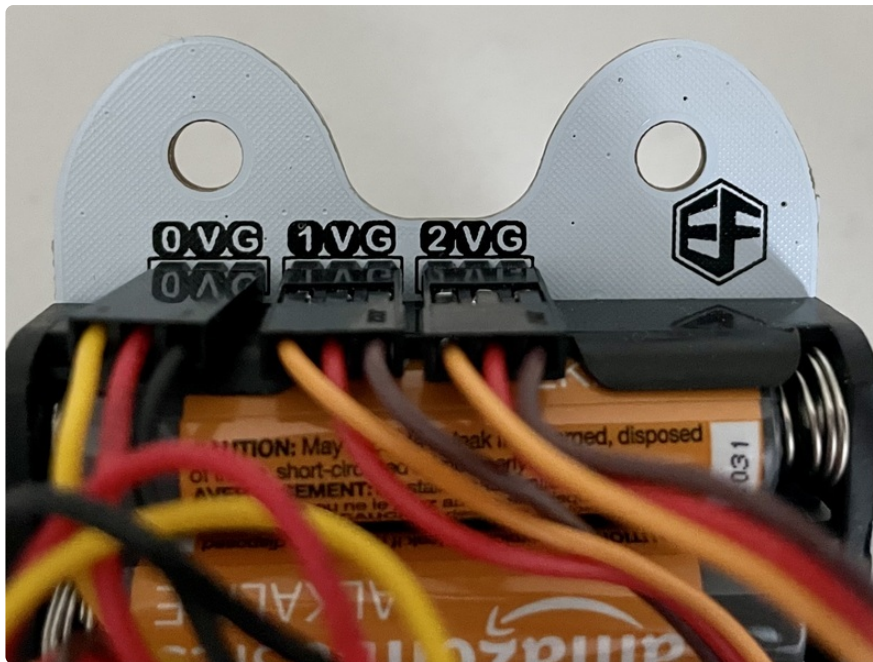


There are a couple of different variations of the board, but both function work without any modifications.



When you get to the part where it asks you to connect a micro:bit, just use the CLUE instead. If you haven't already installed CircuitPython, doing so before connecting it to the car is much easier because it's difficult to reach the reset button. If you have already assembled it, however, with some good timing it's possible to slip a small flathead screw driver under and double press the reset button.

On the back of the board are 3 connectors labeled 0VG through 2VG. This nomenclature indicates the output pins are arranged GPIO (General Purpose Input Output) number, V+, and Ground. As indicated in the photo, the yellow wire should plug towards the GPIO side with the number.



The code is expecting the LED connector to plug into GPIO0, the servo on the robot car's left (when viewed from the rear) to plug into GPIO1, and the right servo should plug into GPIO2. If your robot doesn't move as expected, you can either rearrange the plugs or update it in the code if you'd prefer.

Install Bluefruit Connect App

The Adafruit Bluefruit app is available for both iOS and Android. You can find more information about installing and configuring it in the [Bluefruit LE Connect for iOS and Android \(https://adafru.it/DNc\)](https://adafru.it/DNc) guide. Once that is running, you are ready to continue.

Code the Robot

The code for the robot is divided into two parts. The first part is the main code that is automatically run with CircuitPython. The second part is a custom library for the robot that allows neatly wrapping everything up in a package.

Code and Libraries

After you've finished setting up your CLUE with CircuitPython, you can download all of the code and necessary libraries in the Project Bundle.

To do this, click on the **Download Project Bundle** button in the window below. It will download as a zipped folder. Inside the folder there should be different versions which correspond to the version of CircuitPython you are running.

```
# SPDX-FileCopyrightText: 2022 Melissa LeBlanc-Williams for Adafruit Industries
#
# SPDX-License-Identifier: MIT
import board
```

```

import neopixel
from robot import Robot

# Ring:bit Pins
UNDERLIGHT_PIXELS = board.D0
LEFT_MOTOR = board.D1
RIGHT_MOTOR = board.D2

# Set up the hardware
underlight_neopixels = neopixel.NeoPixel(UNDERLIGHT_PIXELS, 2)
robot = Robot(LEFT_MOTOR, RIGHT_MOTOR, underlight_neopixels)

while True:
    robot.wait_for_connection()
    while robot.is_connected():
        robot.check_for_packets()

```

```

# SPDX-FileCopyrightText: 2022 Melissa LeBlanc-Williams for Adafruit Industries
#
# SPDX-License-Identifier: MIT
import time
import pwmio
import board
import digitalio
import displayio
import vectorio
import terminalio

import neopixel
import adafruit_motor.servo
from adafruit_ble import BLERadio
from adafruit_ble.advertising.standard import ProvideServicesAdvertisement
from adafruit_ble.services.nordic import UARTService
from adafruit_bluefruit_connect.packet import Packet
from adafruit_bluefruit_connect.color_packet import ColorPacket
from adafruit_bluefruit_connect.button_packet import ButtonPacket
from adafruit_display_text.label import Label

# Throttle Directions and Speeds
FWD = 1.0
REV = -1.0
STOP = 0

# Custom Colors
RED = (200, 0, 0)
GREEN = (0, 200, 0)
BLUE = (0, 0, 200)
PURPLE = (120, 0, 160)
YELLOW = (100, 100, 0)
AQUA = (0, 100, 100)

class Robot:
    def __init__(self, left_pin, right_pin, underlight_neopixel):
        self.left_motor = self._init_motor(left_pin)
        self.right_motor = self._init_motor(right_pin)
        self._init_display()
        self._init_ble()
        self.under_pixels = underlight_neopixel
        self.neopixel = neopixel.NeoPixel(board.NEOPIXEL, 1)
        self.direction = STOP
        self.release_color = None
        self.headlights = digitalio.DigitalInOut(board.WHITE_LEDS)
        self.headlights.switch_to_output()
        self.set_underglow(PURPLE)
        self.set_speed(STOP)

    @classmethod

```

```

def _make_palette(cls, color):
    palette = displayio.Palette(1)
    palette[0] = color
    return palette

@classmethod
def _init_motor(cls, pin):
    pwm = pwmio.PWMOut(pin, frequency=50)
    return adafruit_motor.servo.ContinuousServo(pwm, min_pulse=600,
max_pulse=2400)

def _init_ble(self):
    self.ble = BLERadio()
    self.uart_service = UARTService()
    self.advertisement = ProvideServicesAdvertisement(self.uart_service)

def _init_display(self):
    self.display = board.DISPLAY
    self.display_group = displayio.Group()
    self.display.root_group = self.display_group
    self.shape_color = 0
    self.bg_color = 0xFFFF00
    rect = vectorio.Rectangle(
        pixel_shader=self._make_palette(0xFFFF00),
        x=0, y=0,
        width=self.display.width,
        height=self.display.height)
    self.display_group.append(rect)

def wait_for_connection(self):
    self.set_status_led(BLUE)
    self.ble.start_advertising(self.advertisement)
    self._set_status_waiting()
    while not self.ble.connected:
        # Wait for a connection.
        pass
    self.ble.stop_advertising()
    self.set_status_led(GREEN)
    self.set_throttle(STOP)

def is_connected(self):
    return self.ble.connected

def check_for_packets(self):
    if self.uart_service.in_waiting:
        self._process_packet(Packet.from_stream(self.uart_service))

def set_underglow(self, color, save_release_color = False):
    if save_release_color:
        self.release_color = self.get_underglow()
    for index, _ in enumerate(self.under_pixels):
        self.under_pixels[index] = color

def get_underglow(self):
    # Set the 2 Neopixels on the underside fo the robot
    return self.under_pixels[0]

def set_status_led(self, color):
    # Set the status NeoPixel on the CLUE
    self.neopixel[0] = color

def toggle_headlights(self):
    self.headlights.value = not self.headlights.value

def _set_left_throttle(self, speed):
    self.left_motor.throttle = speed

def _set_right_throttle(self, speed):
    # Motor is rotated 180 degrees of the left, so we invert the throttle

```

```

        self.right_motor.throttle = -1 * speed

def rotate_right(self):
    self.release_color = self.get_underglow()
    self.set_underglow(YELLOW, True)
    if self.direction == STOP:
        self._set_status_rotate_cw()
    else:
        self._set_status_right()
        speed = FWD if self.direction == STOP else self.direction
        self._set_left_throttle(speed)
        self._set_right_throttle(STOP if self.direction != STOP else -1 * speed)

def rotate_left(self):
    self.release_color = self.get_underglow()
    self.set_underglow(YELLOW, True)
    if self.direction == STOP:
        self._set_status_rotate_ccw()
    else:
        self._set_status_left()
        speed = FWD if self.direction == STOP else self.direction
        self._set_left_throttle(STOP if self.direction != STOP else -1 * speed)
        self._set_right_throttle(speed)

def set_throttle(self, speed):
    if speed == STOP:
        self._set_status_stop()
    elif speed > STOP:
        self._set_status_forward()
    elif speed < STOP:
        self._set_status_reverse()
    self.set_speed(speed)

def set_speed(self, speed):
    self._set_left_throttle(speed)
    self._set_right_throttle(speed)
    self.direction = speed

def stop(self):
    # Temporarily grab the current color
    color = self.get_underglow()
    self.set_underglow(RED)
    self.set_throttle(STOP)
    time.sleep(0.5)
    self.set_underglow(color)

def _remove_shapes(self):
    while len(self.display_group) > 1:
        self.display_group.pop()

def _add_centered_rect(self, width, height, x_offset=0, y_offset=0, color=None):
    if color is None:
        color = self.shape_color
    rectangle = vectorio.Rectangle(
        pixel_shader=self._make_palette(color),
        width=width,
        height=height,
        x=(self.display.width//2 - width//2) + x_offset - 1,
        y=(self.display.height//2 - height//2) + y_offset - 1
    )
    self.display_group.append(rectangle)

def _add_centered_polygon(self, points, x_offset=0, y_offset=0, color=None):
    if color is None:
        color = self.shape_color
    # Figure out the shape dimensions by using min and max
    width = max(points, key=lambda item:item[0])[0] - min(points, key=lambda
item:item[0])[0]
    height = max(points, key=lambda item:item[1])[1] - min(points, key=lambda

```

```

item:item[1])[1]
    polygon = vectorio.Polygon(
        pixel_shader=self._make_palette(color),
        points=points,
        x=(self.display.width // 2 - width // 2) + x_offset - 1,
        y=(self.display.height // 2 - height // 2) + y_offset - 1
    )
    self.display_group.append(polygon)

def _add_centered_circle(self, radius, x_offset=0, y_offset=0, color=None):
    if color is None:
        color = self.shape_color
    circle = vectorio.Circle(
        pixel_shader=self._make_palette(color),
        radius=radius,
        x=(self.display.width // 2) + x_offset - 1,
        y=(self.display.height // 2) + y_offset - 1
    )
    self.display_group.append(circle)

def _set_status_waiting(self):
    self._remove_shapes()
    text_area = Label(
        terminalio.FONT,
        text="Waiting for\nconnection",
        color=self.shape_color,
        scale=3,
        anchor_point=(0.5, 0.5),
        anchored_position=(self.display.width // 2, self.display.height // 2)
    )
    self.display_group.append(text_area)

def _set_status_reverse(self):
    self._remove_shapes()
    self._add_centered_polygon([(40, 0), (60, 0), (100, 100), (0, 100)], 0, 0)
    self._add_centered_polygon([(0, 40), (100, 40), (50, 0)], 0, -40)

def _set_status_forward(self):
    self._remove_shapes()
    self._add_centered_polygon([(20, 0), (60, 0), (80, 100), (0, 100)])
    self._add_centered_polygon([(0, 0), (150, 0), (75, 50)], 0, 50)

def _set_status_right(self):
    self._remove_shapes()
    self._add_centered_rect(100, 40)
    self._add_centered_polygon([(50, 0), (50, 100), (0, 50)], -50, 0)

def _set_status_rotate_ccw(self):
    self._remove_shapes()
    self._add_centered_circle(80)
    self._add_centered_circle(50, 0, 0, self.bg_color)
    self._add_centered_rect(160, 60, 0, 0, self.bg_color)
    self._add_centered_polygon([(40, 0), (80, 40), (0, 40)], 60, 10)
    self._add_centered_polygon([(40, 40), (80, 0), (0, 0)], -60, -10)

def _set_status_left(self):
    self._remove_shapes()
    self._add_centered_rect(100, 40)
    self._add_centered_polygon([(0, 0), (0, 100), (50, 50)], 50)

def _set_status_rotate_cw(self):
    self._remove_shapes()
    self._add_centered_circle(80)
    self._add_centered_circle(50, 0, 0, self.bg_color)
    self._add_centered_rect(160, 60, 0, 0, self.bg_color)
    self._add_centered_polygon([(40, 0), (80, 40), (0, 40)], -60, 10)
    self._add_centered_polygon([(40, 40), (80, 0), (0, 0)], 60, -10)

def _set_status_stop(self):

```

```

self._remove_shapes()
self._add_centered_rect(100, 100)

def _process_packet(self, packet):
    if isinstance(packet, ColorPacket):
        self._handle_color_packet(packet)
    elif isinstance(packet, ButtonPacket) and packet.pressed:
        # do this when buttons are pressed
        self._handle_button_press_packet(packet)
    elif isinstance(packet, ButtonPacket) and not packet.pressed:
        # do this when some buttons are released
        self._handle_button_release_packet(packet)

def _handle_color_packet(self, packet):
    # Change the color
    self.set_underglow(packet.color)

def _handle_button_press_packet(self, packet):
    if packet.button == ButtonPacket.UP: # UP button pressed
        self.set_throttle(FWD)
    elif packet.button == ButtonPacket.DOWN: # DOWN button
        self.set_throttle(REV)
    elif packet.button == ButtonPacket.RIGHT:
        self.rotate_right()
    elif packet.button == ButtonPacket.LEFT:
        self.rotate_left()
    elif packet.button == ButtonPacket.BUTTON_1:
        self.stop()
    elif packet.button == ButtonPacket.BUTTON_2:
        self.set_underglow(GREEN)
    elif packet.button == ButtonPacket.BUTTON_3:
        self.set_underglow(BLUE)
    elif packet.button == ButtonPacket.BUTTON_4:
        self.toggle_headlights()

def _handle_button_release_packet(self, packet):
    if self.release_color is not None:
        self.set_underglow(self.release_color)
        self.release_color = None
    if packet.button == ButtonPacket.RIGHT:
        self.set_throttle(self.direction)
    if packet.button == ButtonPacket.LEFT:
        self.set_throttle(self.direction)

```

Upload the Code and Libraries to the CLUE

After downloading the Project Bundle, plug your CLUE into the computer's USB port with a known good USB data+power cable. You should see a new flash drive appear in the computer's File Explorer or Finder (depending on your operating system) called **CIRCUITPY**. Unzip the folder and copy the following items to the CLUE's **CIRCUITPY** drive.

- **lib** folder
- **code.py**
- **robot.py**

Your CLUE **CIRCUITPY** drive should look like this after copying the **lib** folder and the **code.py** file.

Code Walkthrough

It starts off by assigning the expected outputs to the board GPIO numbers. For instance, with the `UNDERLIGHT_PIXELS`, it is assigned to D0, which is the same as GPIO 0. Then the libraries are initialized. First the NeoPixel library, then the custom robot library which will be covered below.

```
# Ring:bit Pins
UNDERLIGHT_PIXELS = board.D0
LEFT_MOTOR = board.D1
RIGHT_MOTOR = board.D2

# Set up the hardware
underlight_neopixels = neopixel.NeoPixel(UNDERLIGHT_PIXELS, 2)
robot = Robot(LEFT_MOTOR, RIGHT_MOTOR, underlight_neopixels)
```

The main loop here waits for a connection, then continues to check packets while it is connected. If there is a disconnect, it goes back to waiting for a connection.

```
while True:
    robot.wait_for_connection()
    while robot.is_connected():
        robot.check_for_packets()
```

Robot Custom Library

The robot library is organized so similar functions are grouped together. Near the top, after the imports are done, are a few constants. These are just used to define the directions and speeds. There are also some custom colors set up as tuples arranged in red, green, and blue with values between 0-255.

```
# Throttle Directions and Speeds
FWD = 1.0
REV = -1.0
STOP = 0

# Custom Colors
RED = (200, 0, 0)
GREEN = (0, 200, 0)
BLUE = (0, 0, 200)
PURPLE = (120, 0, 160)
YELLOW = (100, 100, 0)
AQUA = (0, 100, 100)
```

The rest of the code is inside of the `Robot` class itself. The first part initializes the robot hardware by calling several initialization functions that will be covered below. Initial variable values are set, and the car is put into a `STOP` state with the underglow set to the value of `PURPLE`.

```

def __init__(self, left_pin, right_pin, underlight_neopixel):
    self.left_motor = self._init_motor(left_pin)
    self.right_motor = self._init_motor(right_pin)
    self._init_display()
    self._init_ble()
    self.under_pixels = underlight_neopixel
    self.neopixel = neopixel.NeoPixel(board.NEOPIXEL, 1)
    self.direction = STOP
    self.release_color = None
    self.headlights = digitalio.DigitalInOut(board.WHITE_LEDS)
    self.headlights.switch_to_output()
    self.set_underglow(PURPLE)
    self.set_speed(STOP)

```

In the next section, there are a couple of functions with the `@classmethod` decorator. This allows the functions to be part of the class without actually making use of the `self` variable.

The first function just wraps takes the given color and assigns it to a `displayio.Palette` object and returns that object.

The other three functions just initialize the given motor, BLE (or Bluetooth Low Energy), and the Display.

Note that any functions that start with the underscore character "`_`" are considered private functions by Python and not intended to be called from outside of the class.

```

@classmethod
def _make_palette(cls, color):
    palette = displayio.Palette(1)
    palette[0] = color
    return palette

@classmethod
def _init_motor(cls, pin):
    pwm = pwmio.PWMOut(pin, frequency=50)
    return adafruit_motor.servo.ContinuousServo(pwm, min_pulse=600, max_pulse=2400)

def _init_ble(self):
    self.ble = BLERadio()
    self.uart_service = UARTService()
    self.advertisement = ProvideServicesAdvertisement(self.uart_service)

def _init_display(self):
    self.display = board.DISPLAY
    self.display_group = displayio.Group()
    self.display.show(self.display_group)
    self.shape_color = 0
    self.bg_color = 0xFFFF00
    rect = vectorio.Rectangle(
        pixel_shader=self._make_palette(0xFFFF00),
        x=0, y=0,
        width=self.display.width,
        height=self.display.height)
    self.display_group.append(rect)

```

Next are a few functions that are used in the main program loop to manage the BLE connection properly as well as check if any new packets have been sent from the Bluefruit Connect app and send all of those to the `_process_packet()` function.

```
def wait_for_connection(self):
    self.set_status_led(BLUE)
    self.ble.start_advertising(self.advertisement)
    self._set_status_waiting()
    while not self.ble.connected:
        # Wait for a connection.
        pass
    self.ble.stop_advertising()
    self.set_status_led(GREEN)
    self.set_throttle(STOP)

def is_connected(self):
    return self.ble.connected

def check_for_packets(self):
    if self.uart_service.in_waiting:
        self._process_packet(Packet.from_stream(self.uart_service))
```

These function are for controlling the NeoPixels and the LED on the CLUE. The `get_underglow()` function saves a little memory by making use of the values that are stored in the NeoPixel library to get the current color.

```
def set_underglow(self, color, save_release_color = False):
    if save_release_color:
        self.release_color = self.get_underglow()
    for index, _ in enumerate(self.under_pixels):
        self.under_pixels[index] = color

def get_underglow(self):
    # Set the 2 Neopixels on the underside fo the robot
    return self.under_pixels[0]

def set_status_led(self, color):
    # Set the status NeoPixel on the CLUE
    self.neopixel[0] = color

def toggle_headlights(self):
    self.headlights.value = not self.headlights.value
```

The next set of functions are for controlling the movement of the robot. Servos spin in the same direction, but because they are mounted at 180 degrees to each other, one wheel wants to go forward and the other backwards. It starts out with the left and right throttle functions which solves this.

The rotate functions will behave a little different depending on whether the robot is currently stopped or currently moving in a direction. If it is stopped, the robot will rotate in place, whereas either the left or right wheel will act as a pivot if the robot is moving forward or backward.

The `set_throttle()` and `set_speed()` functions are similar except that the `set_throttle` will handle updating the display and underglow in addition to setting the speed.

The `stop()` function will also set the underglow to the value of `RED` for half a second after stopping to simulate brake lights before returning to the previously set color.

```
def _set_left_throttle(self, speed):
    self.left_motor.throttle = speed

def _set_right_throttle(self, speed):
    # Motor is rotated 180 degrees of the left, so we invert the throttle
    self.right_motor.throttle = -1 * speed

def rotate_right(self):
    self.release_color = self.get_underglow()
    self.set_underglow(YELLOW, True)
    if self.direction == STOP:
        self._set_status_rotate_cw()
    else:
        self._set_status_right()
    speed = FWD if self.direction == STOP else self.direction
    self._set_left_throttle(speed)
    self._set_right_throttle(STOP if self.direction != STOP else -1 * speed)

def rotate_left(self):
    self.release_color = self.get_underglow()
    self.set_underglow(YELLOW, True)
    if self.direction == STOP:
        self._set_status_rotate_ccw()
    else:
        self._set_status_left()
    speed = FWD if self.direction == STOP else self.direction
    self._set_left_throttle(STOP if self.direction != STOP else -1 * speed)
    self._set_right_throttle(speed)

def set_throttle(self, speed):
    if speed == STOP:
        self._set_status_stop()
    elif speed > STOP:
        self._set_status_forward()
    elif speed < STOP:
        self._set_status_reverse()
    self.set_speed(speed)

def set_speed(self, speed):
    self._set_left_throttle(speed)
    self._set_right_throttle(speed)
    self.direction = speed

def stop(self):
    # Temporarily grab the current color
    color = self.get_underglow()
    self.set_underglow(RED)
    self.set_throttle(STOP)
    time.sleep(0.5)
    self.set_underglow(color)
```

Next are the drawing helpers. The first one removes everything from the main `displayio` group and acts much like `clear()` function.

The remain three function will draw at the center of the display and offset from there if offsets are provided. The `_add_centered_polygon()` function makes use of a lambda to return only the x and y values, so it can figure out the largest and smallest points on the respective axis to get the overall geometry of the shape for easy centering.

Once the absolute coordinates are determined, everything is passed to the **vectorio** shape drawing functions.

```
def _remove_shapes(self):
    while len(self.display_group) > 1:
        self.display_group.pop()

def _add_centered_rect(self, width, height, x_offset=0, y_offset=0, color=None):
    if color is None:
        color = self.shape_color
    rectangle = vectorio.Rectangle(
        pixel_shader=self._make_palette(color),
        width=width,
        height=height,
        x=(self.display.width//2 - width//2) + x_offset - 1,
        y=(self.display.height//2 - height//2) + y_offset - 1
    )
    self.display_group.append(rectangle)

def _add_centered_polygon(self, points, x_offset=0, y_offset=0, color=None):
    if color is None:
        color = self.shape_color
    # Figure out the shape dimensions by using min and max
    width = max(points, key=lambda item:item[0])[0] - min(points, key=lambda
item:item[0])[0]
    height = max(points, key=lambda item:item[1])[1] - min(points, key=lambda
item:item[1])[1]
    polygon = vectorio.Polygon(
        pixel_shader=self._make_palette(color),
        points=points,
        x=(self.display.width // 2 - width // 2) + x_offset - 1,
        y=(self.display.height // 2 - height // 2) + y_offset - 1
    )
    self.display_group.append(polygon)

def _add_centered_circle(self, radius, x_offset=0, y_offset=0, color=None):
    if color is None:
        color = self.shape_color
    circle = vectorio.Circle(
        pixel_shader=self._make_palette(color),
        radius=radius,
        x=(self.display.width // 2) + x_offset - 1,
        y=(self.display.height // 2) + y_offset - 1
    )
    self.display_group.append(circle)
```

The next group of functions make use of the shape drawing functions to draw specific graphics. The only exception to this is the `_set_status_waiting()` function, which simply creates a label.

```
def _set_status_waiting(self):
    self._remove_shapes()
    text_area = Label(
        terminalio.FONT,
```

```

        text="Waiting for\nconnection",
        color=self.shape_color,
        scale=3,
        anchor_point=(0.5, 0.5),
        anchored_position=(self.display.width // 2, self.display.height // 2)
    )
    self.display_group.append(text_area)

def _set_status_reverse(self):
    self._remove_shapes()
    self._add_centered_polygon([(40, 0), (60, 0), (100, 100), (0, 100)], 0, 0)
    self._add_centered_polygon([(0, 40), (100, 40), (50, 0)], 0, -40)

def _set_status_forward(self):
    self._remove_shapes()
    self._add_centered_polygon([(20, 0), (60, 0), (80, 100), (0, 100)])
    self._add_centered_polygon([(0, 0), (150, 0), (75, 50)], 0, 50)

def _set_status_right(self):
    self._remove_shapes()
    self._add_centered_rect(100, 40)
    self._add_centered_polygon([(50, 0), (50, 100), (0, 50)], -50, 0)

def _set_status_rotate_ccw(self):
    self._remove_shapes()
    self._add_centered_circle(80)
    self._add_centered_circle(50, 0, 0, self.bg_color)
    self._add_centered_rect(160, 60, 0, 0, self.bg_color)
    self._add_centered_polygon([(40, 0), (80, 40), (0, 40)], 60, 10)
    self._add_centered_polygon([(40, 40), (80, 0), (0, 0)], -60, -10)

def _set_status_left(self):
    self._remove_shapes()
    self._add_centered_rect(100, 40)
    self._add_centered_polygon([(0, 0), (0, 100), (50, 50)], 50)

def _set_status_rotate_cw(self):
    self._remove_shapes()
    self._add_centered_circle(80)
    self._add_centered_circle(50, 0, 0, self.bg_color)
    self._add_centered_rect(160, 60, 0, 0, self.bg_color)
    self._add_centered_polygon([(40, 0), (80, 40), (0, 40)], -60, 10)
    self._add_centered_polygon([(40, 40), (80, 0), (0, 0)], 60, -10)

def _set_status_stop(self):
    self._remove_shapes()
    self._add_centered_rect(100, 100)

```

Finally, there are the functions that actually handle the packets. If you would like to modify what the buttons in the Bluefruit app do, this is where you would change that.

The first function will take the given packet and if it is one of the expected packet types, it will route it to the appropriate handler function, otherwise it will be ignored.

The handler functions then look at more specific details in the packet and call the function for whatever action should be performed.

```

def _process_packet(self, packet):
    if isinstance(packet, ColorPacket):
        self._handle_color_packet(packet)
    elif isinstance(packet, ButtonPacket) and packet.pressed:
        # do this when buttons are pressed
        self._handle_button_press_packet(packet)

```

```

        elif isinstance(packet, ButtonPacket) and not packet.pressed:
            # do this when some buttons are released
            self._handle_button_release_packet(packet)

def _handle_color_packet(self, packet):
    # Change the color
    self.set_underglow(packet.color)

def _handle_button_press_packet(self, packet):
    if packet.button == ButtonPacket.UP: # UP button pressed
        self.set_throttle(FWD)
    elif packet.button == ButtonPacket.DOWN: # DOWN button
        self.set_throttle(REV)
    elif packet.button == ButtonPacket.RIGHT:
        self.rotate_right()
    elif packet.button == ButtonPacket.LEFT:
        self.rotate_left()
    elif packet.button == ButtonPacket.BUTTON_1:
        self.stop()
    elif packet.button == ButtonPacket.BUTTON_2:
        self.set_underglow(GREEN)
    elif packet.button == ButtonPacket.BUTTON_3:
        self.set_underglow(BLUE)
    elif packet.button == ButtonPacket.BUTTON_4:
        self.toggle_headlights()

def _handle_button_release_packet(self, packet):
    if self.release_color is not None:
        self.set_underglow(self.release_color)
        self.release_color = None
    if packet.button == ButtonPacket.RIGHT:
        self.set_throttle(self.direction)
    if packet.button == ButtonPacket.LEFT:
        self.set_throttle(self.direction)

```

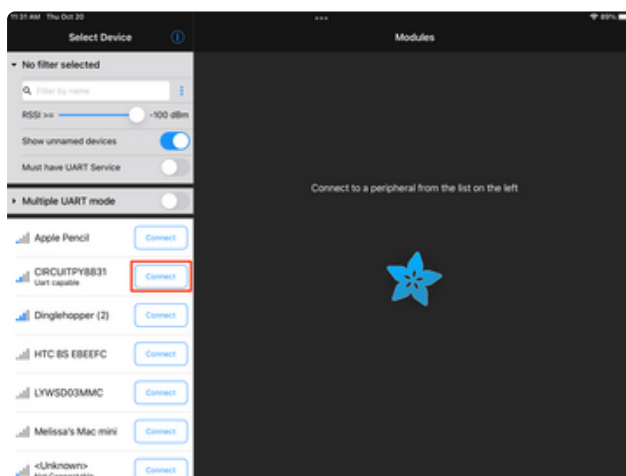
Usage

To use the robot, start by switching it on. There's a small switch just under the left side of the CLUE. Once it has started up, a "Waiting for connection" message will appear. This means it is listening for the Bluefruit App.

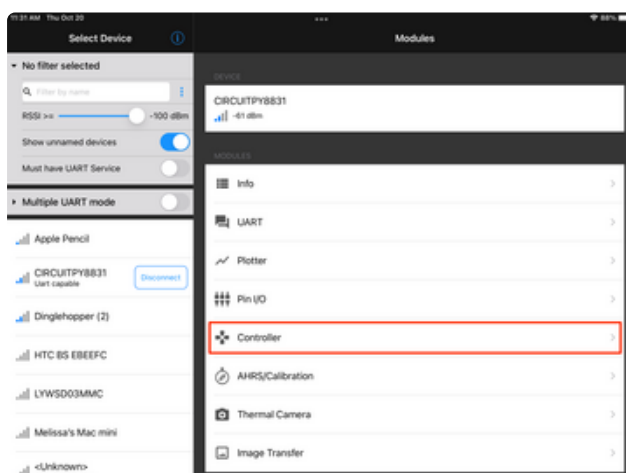


Connect with Bluefruit App

On your mobile device, open the Bluefruit Connect App. You should see a list of devices. If you are using a tablet or similar device with a larger screen, the list will appear on the left side.



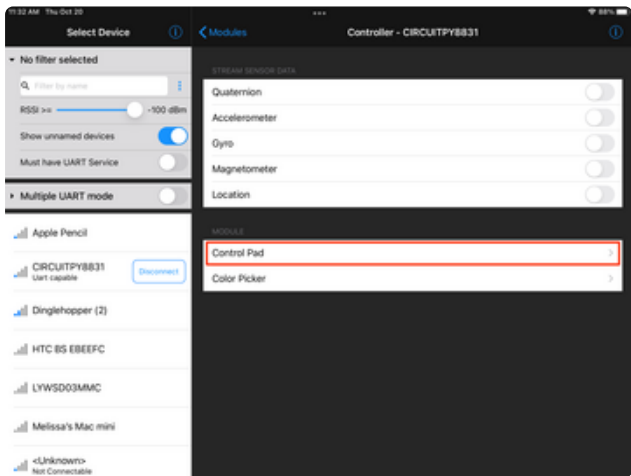
You should see a device that begins with CIRCUITPY in the list. Click the Connect button next to it.



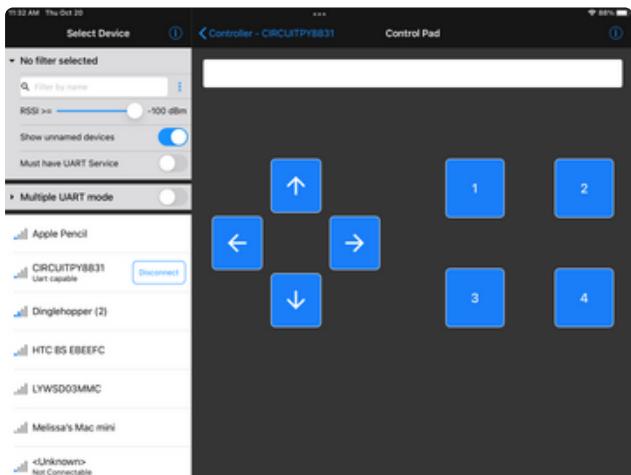
Once you have connected, you should see a screen similar to this. Everything you need is under the **Controller** option, so select that.

Driving the Robot

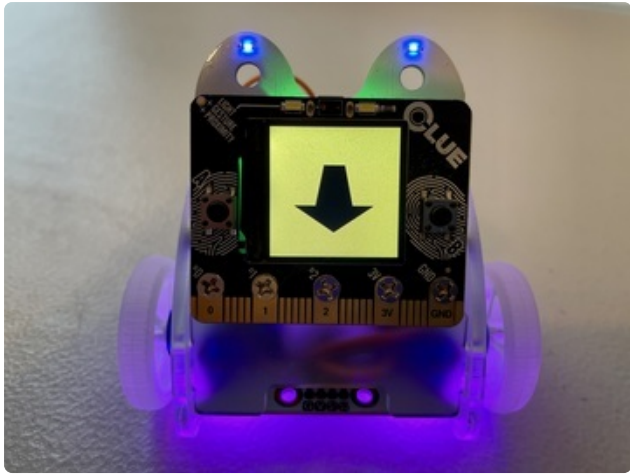
To drive the robot around, you will want to use the Control Pad. This gives you 4 directions plus 4 auxiliary function buttons.



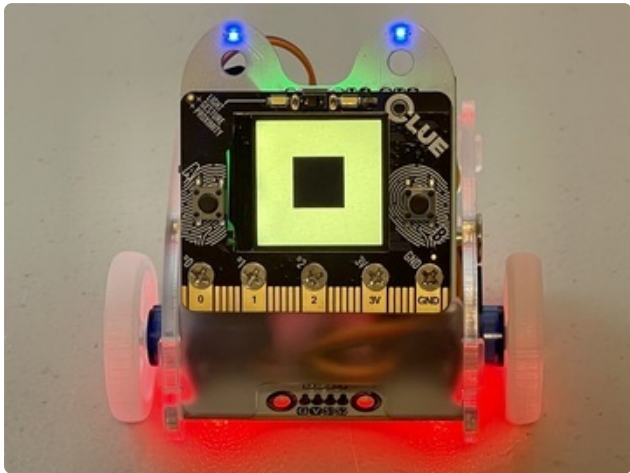
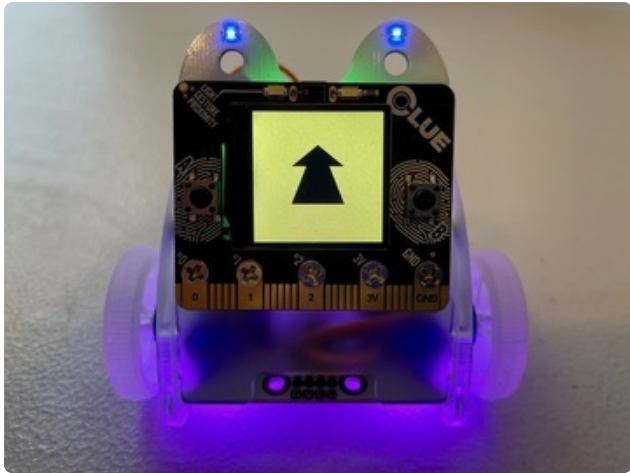
Select the **Control Pad** option, which is under Controller. Press the back button on your device first if you're in the Color Picker.



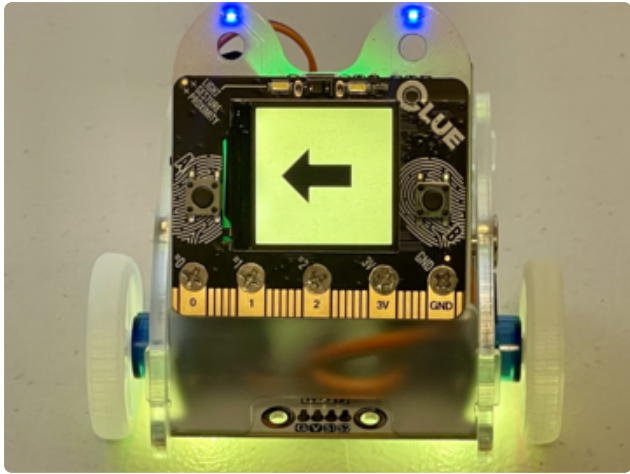
You will be presented with a control pad with directional and function buttons.



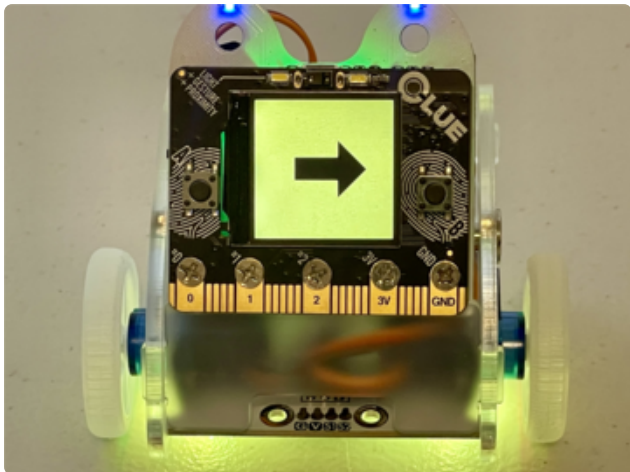
Pressing **Up** starts the robot moving forward and **Down** starts it moving Backwards. The image will change to show the direction the robot is moving.



Pressing **Button 1** will stop the robot. It will light up red for a brief period to simulate brake lights.

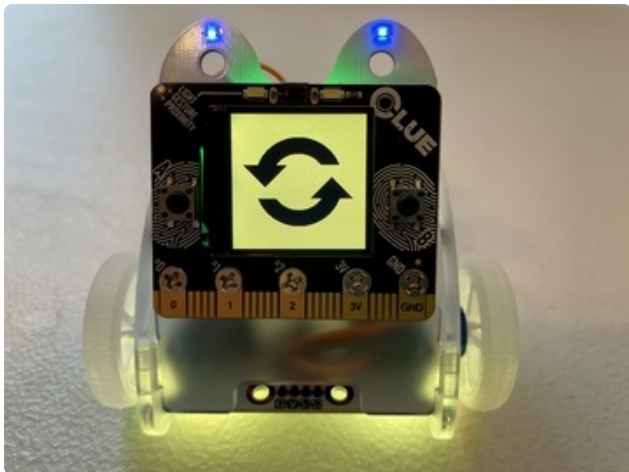


While the robot is moving forward or backward, holding either the **Left** or **Right** button will cause it to turn in that direction. Additionally, it will light up yellow and a side arrow will appear on the display that points in the direction it is turning.





If the robot is stopped, it will rotate in place. It will also light up yellow and a pair of circular arrows will appear on the display that indicate the direction it is rotating.



Turning On and Off Lights

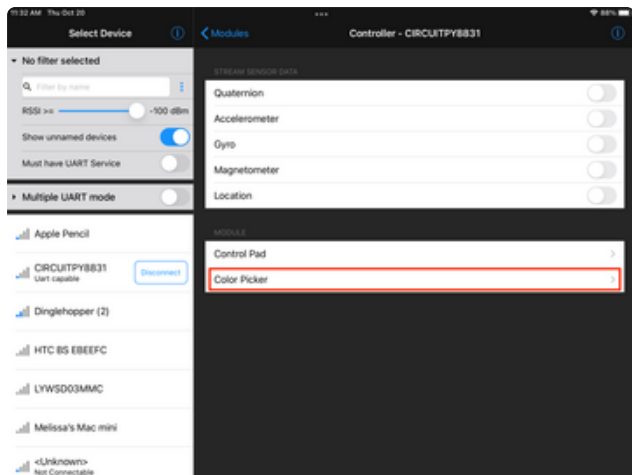


To turn the lights on or off, make sure you are in the control pad and just press button 4 to toggle them.

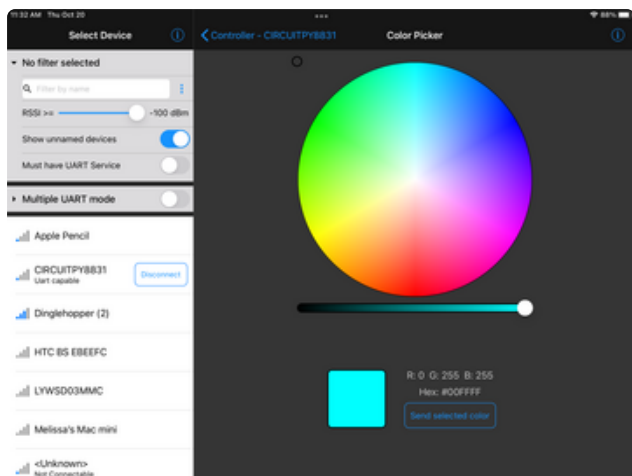
Changing Underglow Color

There are a couple of ways to change the underglow color. A couple of the Control Pad function buttons have preset colors. Button 2 will change it to **blue** and button 3 will change it to **green**.

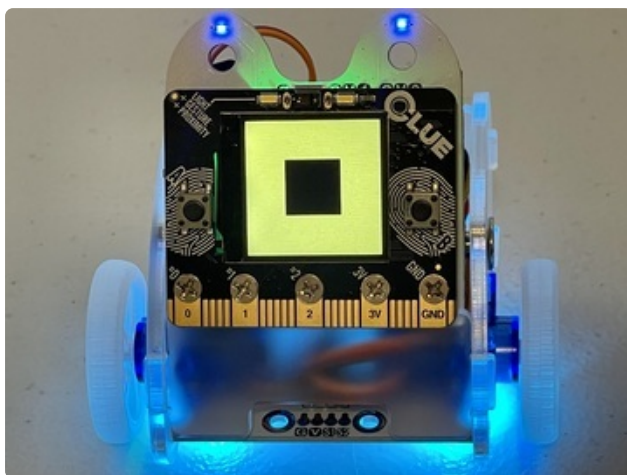
You can also set it to any color you like with the color picker.



Select the **Color Picker** option, which is under Controller. Press the back button on your device first if you're currently in the Control Pad.



Just select the color you would like to use and press the **Send Selected Color** button to set the new underglow color.



The lights underneath should now be glowing the selected color.

Adding More Sensors

There are plenty of sensors on the CLUE that could also be used such as a 9-DoF motion sensor, light sensor, sound sensor, and even a temperature/humidity sensor. You could also make use of the buttons on the CLUE to change modes and have the controller buttons do something completely different in the new mode.