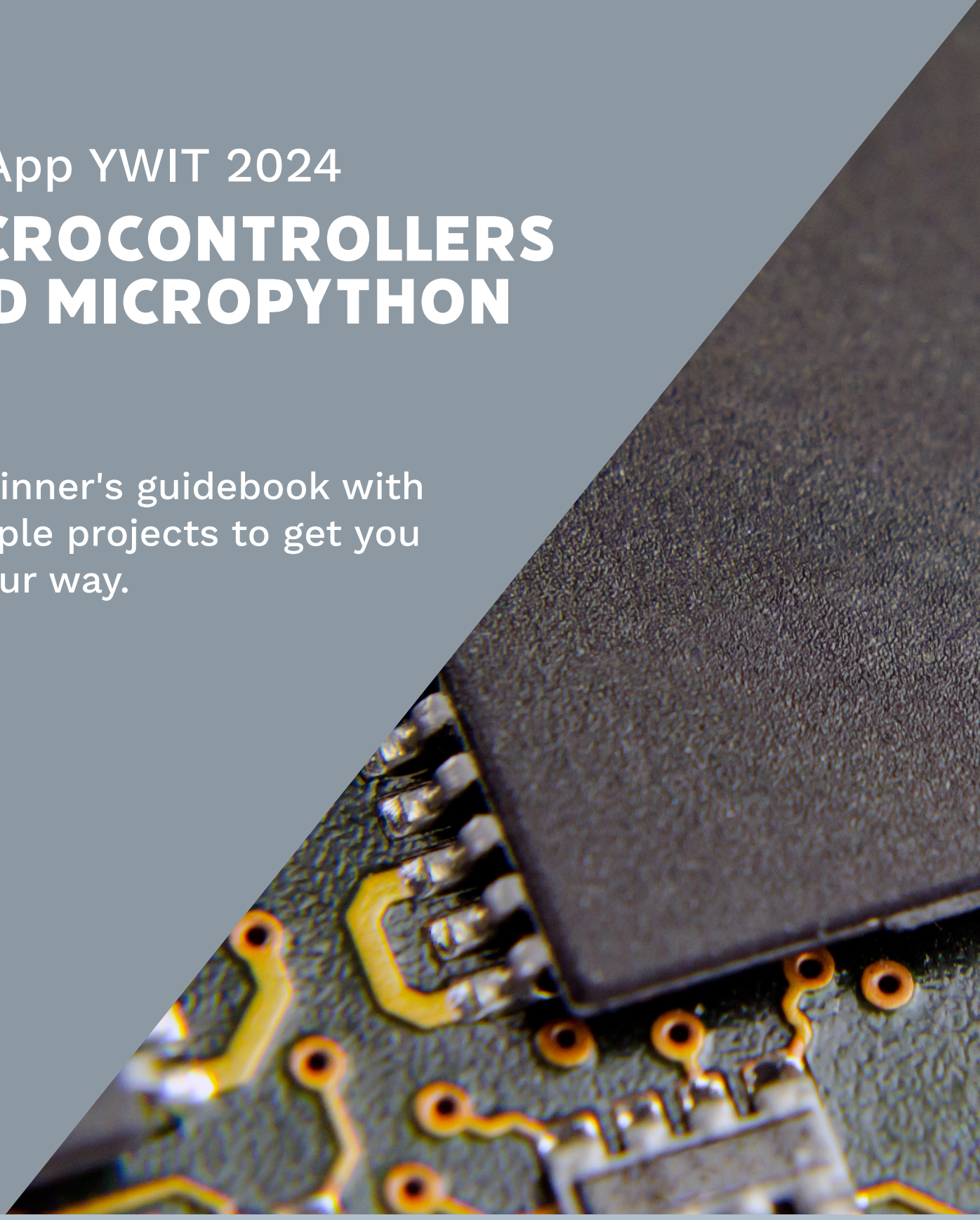
NetApp YWIT 2024

MICROCONTROLLERS AND MICROPYTHON

A beginner's guidebook with 7 simple projects to get you on your way.



September 26, 2024

Contents

Chapter 1: Introduction

1.1 What this course will teach

This workshop will introduce the student to Python coding, electronics, and project design. We will be building several projects ranging from simple (less components and simpler code) to more complex (more components or more complicated code). These projects are based on the ESP32-C3 microcontroller (specifically the development board produced by Seeed Studio) which is running MicroPython and they depend on some other electronics components such as LEDs, buttons, and more.

Students are encouraged to make use of the written instructions and diagrams and to explore and play with the components for themselves to see how they work and what they do. There are many online resources as well for learning.

1.2 Working with the microcontroller

1.2.1 Installing the driver

Connecting the microcontroller to your computer may require the installation of a USB driver. If so, download the driver for your OS from this page and install as instructed: https://ftdichip.com/drivers/vcp-drivers/. Once installed successfully, unplug and replug the USB cable into the microcontroller and then press the small reset button on the microcontroller board labeled "R" (for reset).

1.2.2 Pinout Reference

On any integrated circuit, that is a chip that contains electronic logic, a pinout is a diagram and description of what each of the pins of that circuit are used for. This is a very useful document for any electronics engineer to have handy and to reference. The pinout diagram for the microcontroller that we are using as part of this workshop is replicated below:

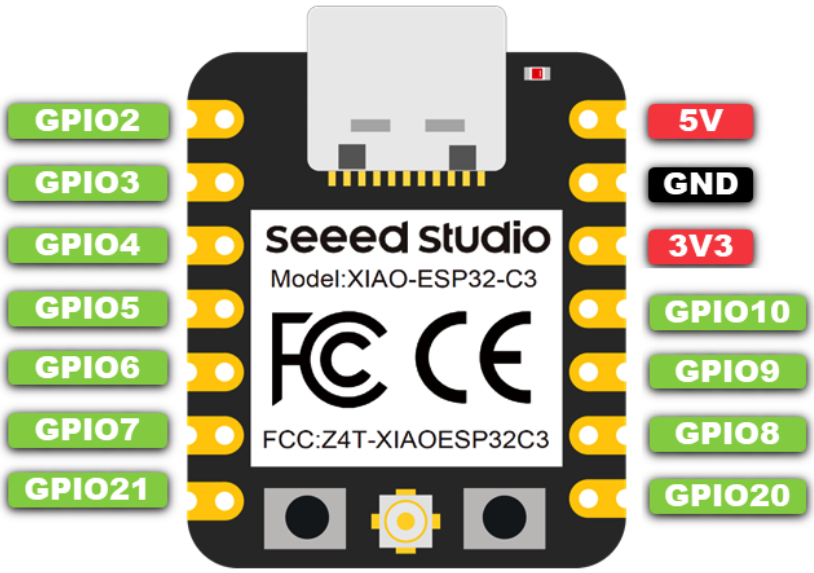


Figure 1.1: A simplified pinout of the Seeed Studio ESP32-C3 development board

In this diagram, you'll notice several things:

- An image of the microcontroller board in the center
- A label for each pin indicating its function

In the code throughout this guide, you will see references to pin numbers like **Pin(5)**. Wherever you see this, it means that the code is expecting a connection to, in this example, the 4th pin down on the left. If you find that your code is not working as expected, double check the pin numbers in the code vs. the placement of your wires.

Chapter 2: Electronics Essentials

Electronics is a facinating world. This beginner's guide will introduce you to the essential components and tools you'll need to start building your own simple circuits. We'll cover the basics of breadboards, LEDs, buttons, resistors, and jumper cables. You should feel free to explore on your own as well using the components in your kit. While this guide will be useful, it is not even close to everything you'll need to know. There is always more to learn and if electronics interests you, there are lots of online tutorials as well as formal education to guide you further.

2.1 The Breadboard: Your Circuit's Foundation

2.1.1 What is a breadboard?

A breadboard is a reusable platform for prototyping electronic circuits without the need for soldering. It's a plastic board with numerous tiny holes arranged in rows and columns. These holes are connected internally, making it easy to connect components together.

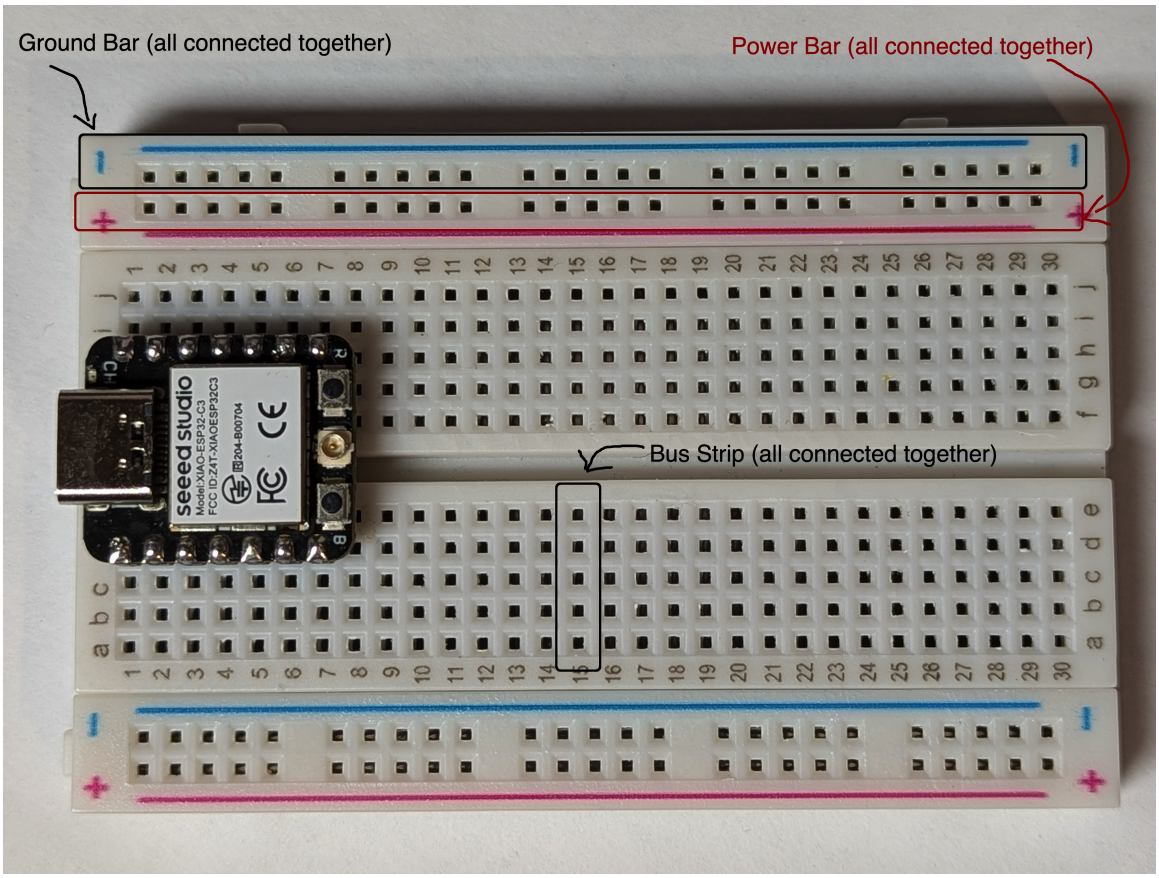
2.1.2 Understanding the layout

Power rails

The long horizontal rows along the top and bottom edges are typically used as power rails to distribute power (positive and negative) throughout the circuit.

Internal connections

The remaining rows are divided into vertical columns, with each column having five holes connected internally. These connections allow you to easily connect multiple components within the same column.



2.2 LEDs: Lighting Up Your Circuits

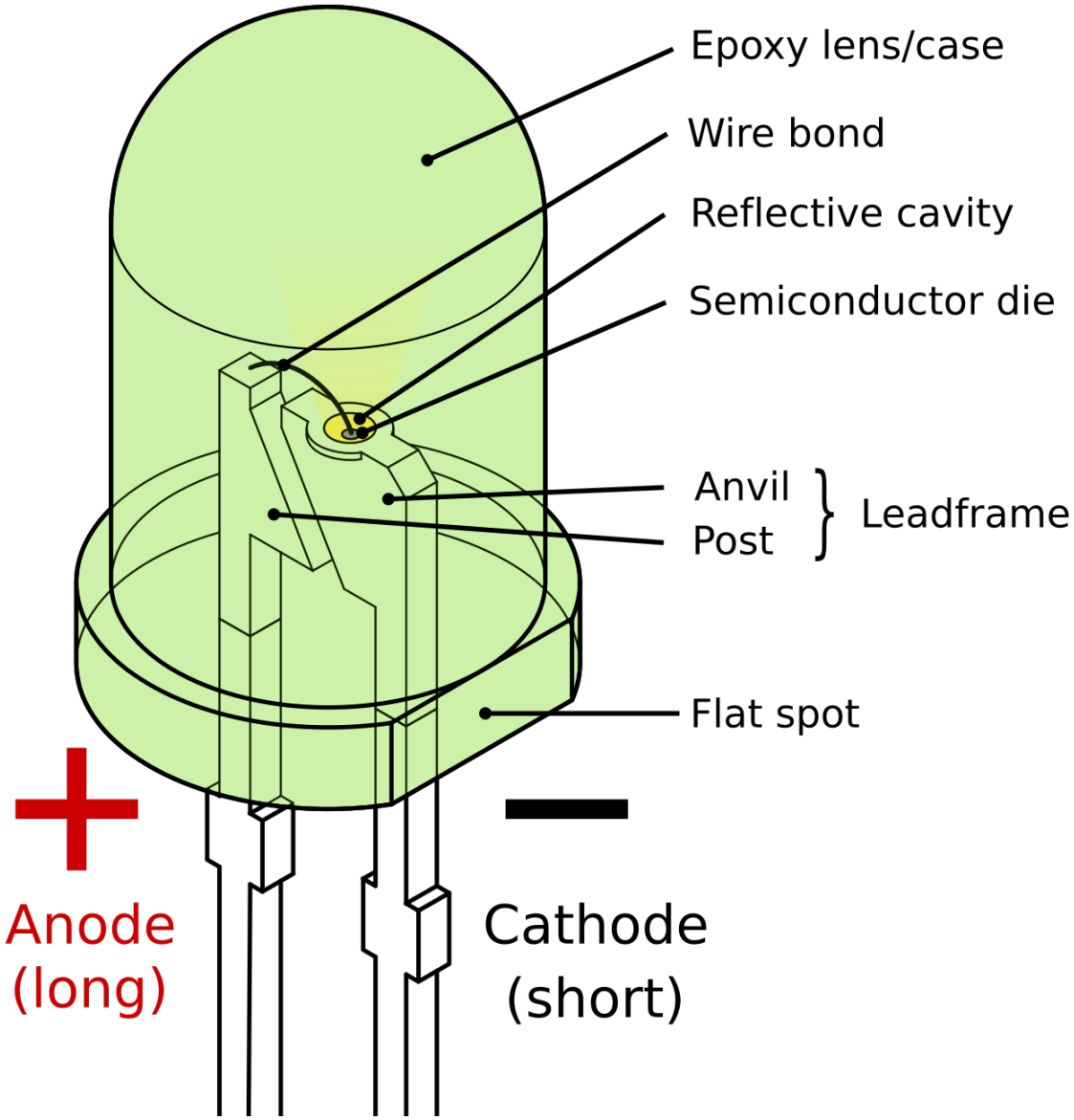
2.2.1 What is an LED?

An LED (Light-Emitting Diode) is a semiconductor device that emits light when an electric current passes through it. They are available in various colors and are commonly used as indicators or displays in electronic circuits.

2.2.2 Polarity

LEDs have two leads:

- **Anode (longer lead):** Connected to the positive (+) side of the power supply.
- **Cathode (shorter lead):** Connected to the negative (-) side of the power supply.



2.2.3 Current limiting resistor

It's crucial to connect a resistor in series with an LED to limit the current flowing through it. Otherwise, the LED may get damaged or burn out. The value of the resistor depends on the LED's specifications and the power supply voltage.

2.3 Buttons: Interacting with Your Circuits

2.3.1 What is a button?

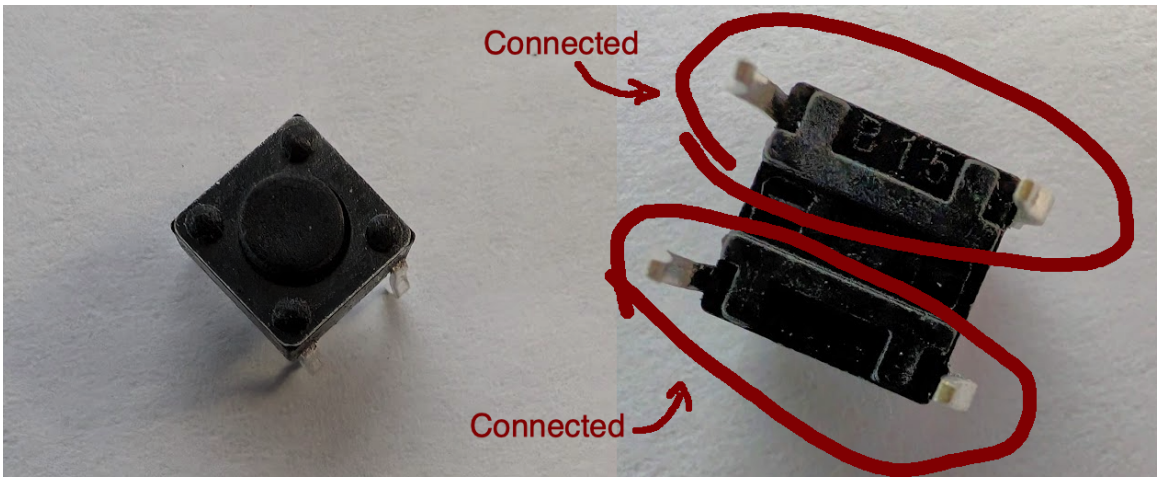
A button, also known as a pushbutton switch, is a simple mechanical device that completes or breaks an electrical circuit when pressed. They are commonly used to trigger actions or provide input to electronic circuits.

2.3.2 Types of buttons

- **Normally open (NO):** The circuit is open (no connection) when the button is not pressed and closes (connection made) when pressed.
- **Normally closed (NC):** The circuit is closed (connection made) when the button is not pressed and opens (no connection) when pressed.

2.3.3 Connected Pins

Many buttons will have more than two pins (often 4). If they do, then often you can see on the bottom that some of those pins will already be connected together. Pay attention to this so that you wire it up the right way.



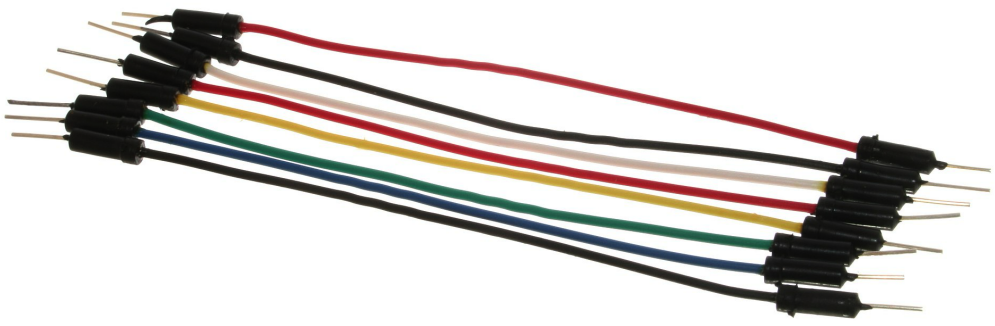
2.4 Jumper Cables: Making Connections

2.4.1 What are jumper cables?

Jumper cables are short wires with connectors at both ends used to make temporary connections between components on a breadboard or between a breadboard and other devices.

2.4.2 Using jumper cables

Insert the connectors into the appropriate holes on the breadboard or device to establish a connection.



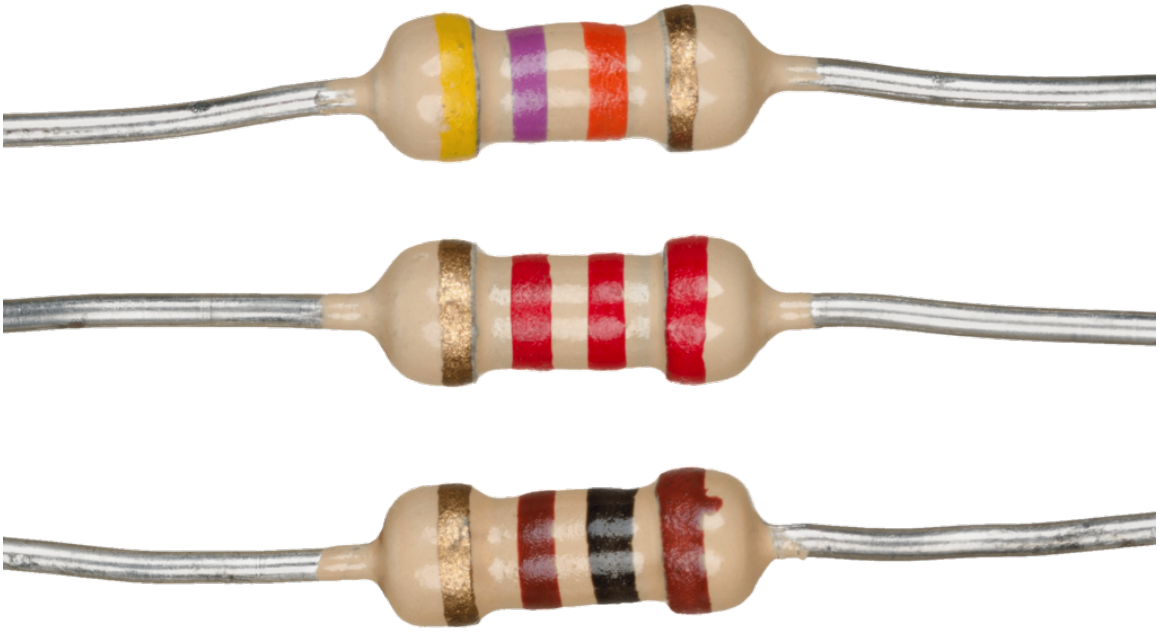
2.5 Resistors: Controlling Current Flow

2.5.1 What is a resistor?

A resistor is a passive component that limits the flow of electric current in a circuit. They are essential for protecting components (like LEDs) from excessive current and for controlling voltage levels in circuits.

2.5.2 Resistance value

The resistance value is measured in ohms (Ω) and is indicated by colored bands on the resistor body. You can use a resistor color code chart to decode the value.



2.6 Safety Tips:

- Always disconnect the power supply before making any changes to the circuit.
- Be careful not to short-circuit the power supply by connecting the positive and negative rails directly.
- If you smell burning or see smoke, immediately disconnect the power supply.

Chapter 3: MicroPython IDE

Throughout this guide, we will be making use of a free online IDE for connecting our microcontroller to our computer and loading, editing, and running code on it. You can access the IDE from your browser by going to https://viper-ide.org/. Click on the USB icon in the top right and choose your microcontroller from the list:

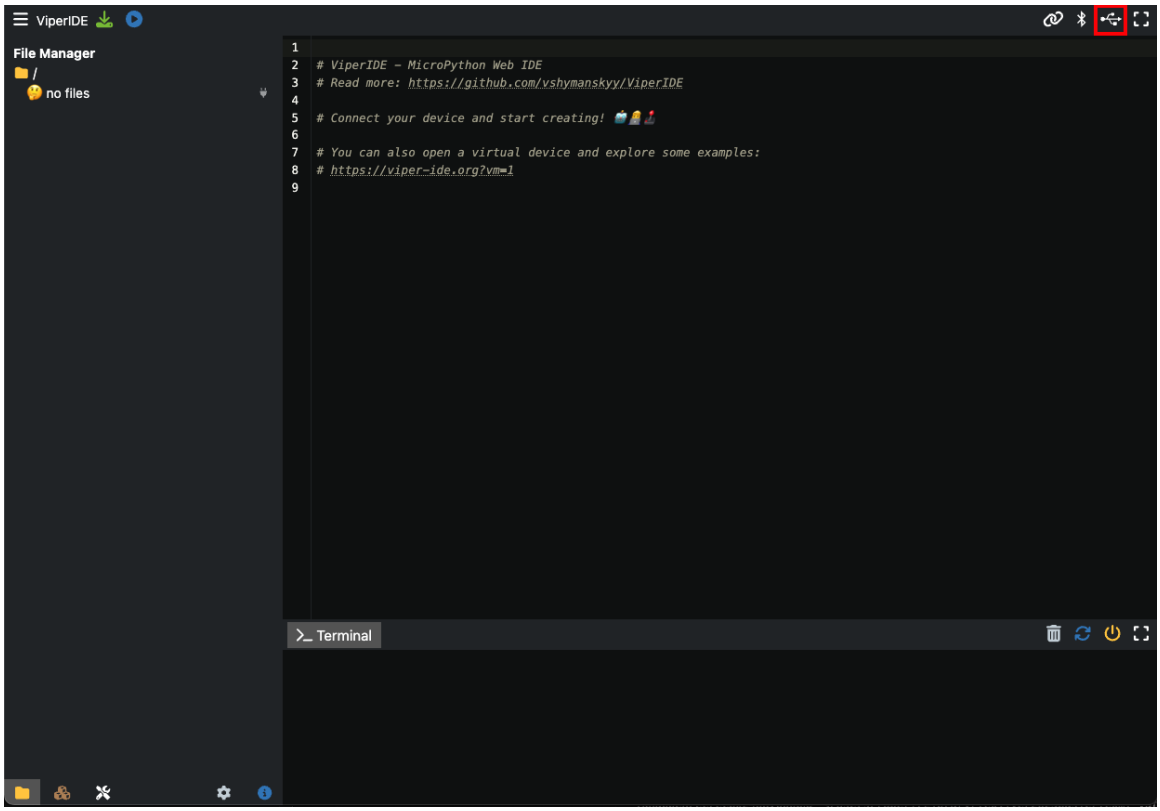


Figure 3.1: Click the button highlighted in red.

If you see multiple items in the dialog that pops up, choose the one that starts with "USB JTAG". See below for an example:

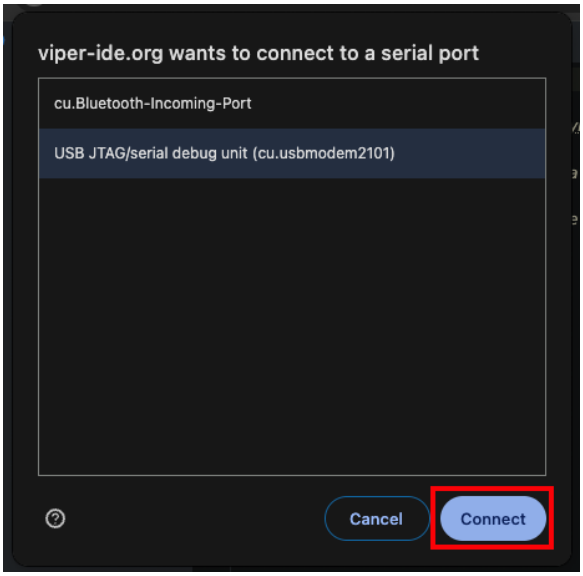


Figure 3.2: Click the button highlighted in red.

Once you have connected, you will see a green dialog labeled "Device connected" and the file manager on the list will populate with the list of files installed on the device:

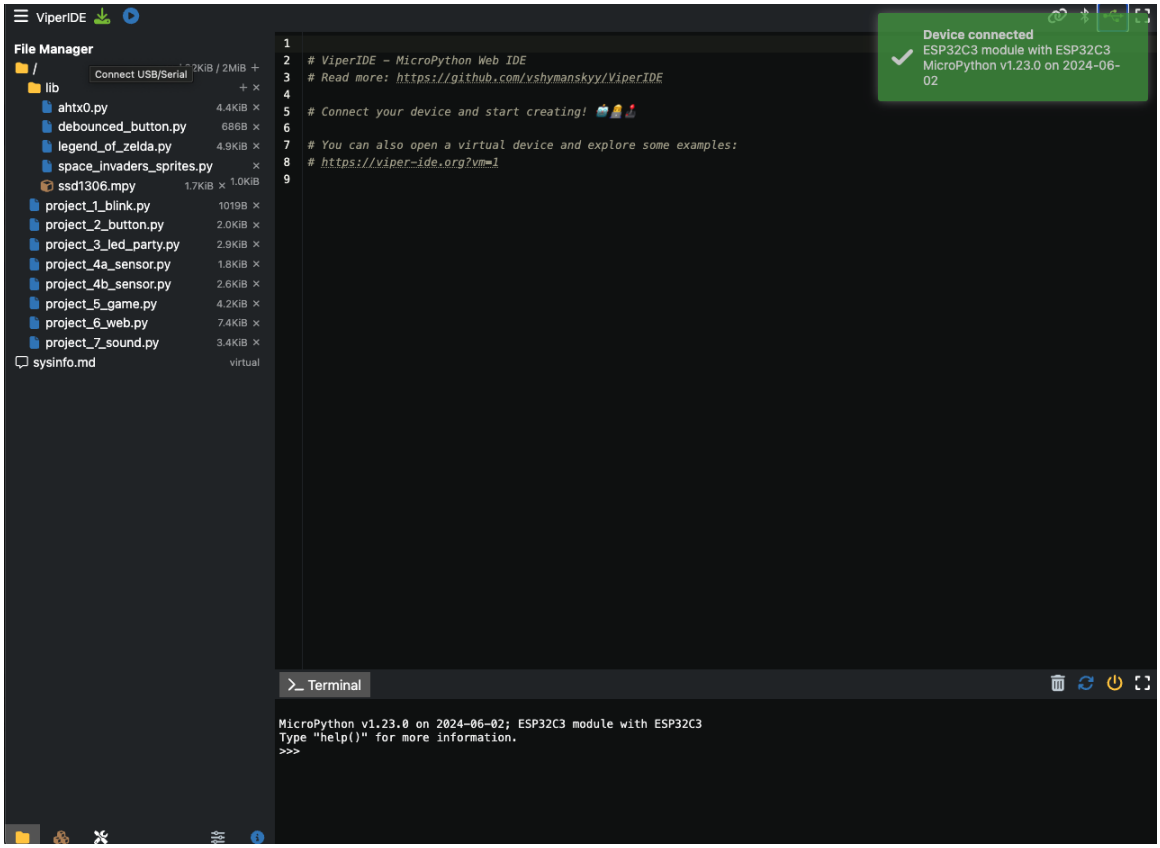


Figure 3.3: Make sure to open the right file for this project

After your device is connected, refer back to the chapter you are working on for the next steps.

Chapter 4: Project 1: Blink

4.1 Overview

This project is designed to provide a foundation for subsequent projects in this book (**and beyond**). Over the course of this project, you will:

- Create a simple circuit using your breadboard
- Write a program that runs in a loop
- Use MicroPython in your program to interact with your microcontroller's GPIO pins.

At the end of this project, your microcontroller should run a MicroPython program which alternates a light between its ON and OFF states. Let's get started!

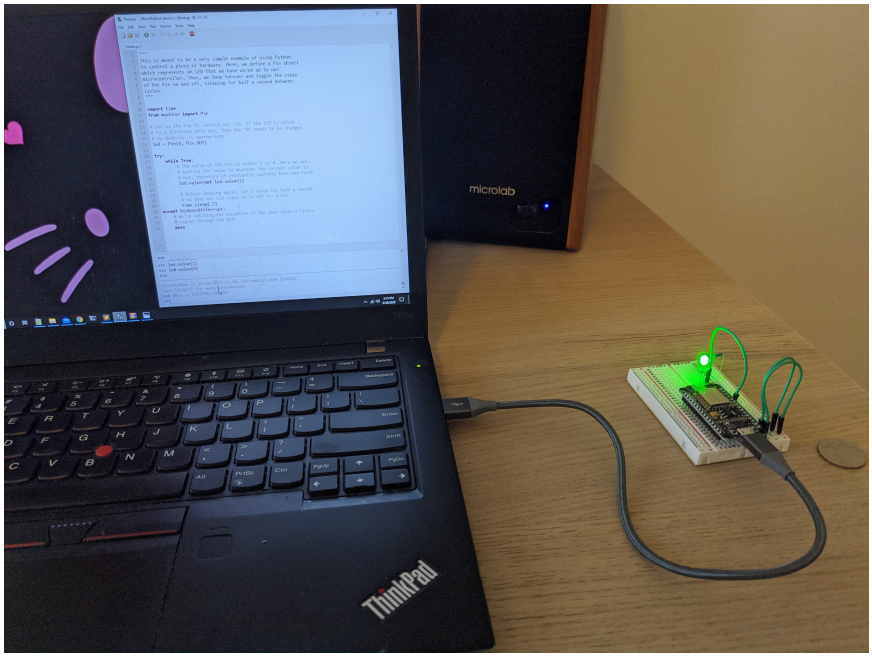


Figure 4.1: The end result should look something like this

4.2 Directions

4.2.1 Creating the circuit

Using jumper cables, you will be assembling a circuit between your microcontroller, your breadboard, an LED, and a 220Ω resistor.

Attach the microcontroller to the breadboard

Carefully insert the pins at the bottom of your microcontroller into the breadboard, making sure that the microcontroller is oriented such that:

- The pin labeled **3V3** is inserted in hole at **Column C, Row 1** of the breadboard (or **C1**, for short)
- The pin labeled **Vin** is inserted in hole **J1** of the breadboard
- The pin labeled **D0** is inserted in hole **C15** of the breadboard
- the pin labeled **A0** is inserted in hole **J15** of the breadboard

You may need to apply more pressure than expected to seat the microcontroller properly in the breadboard. When its over, it should look like this:

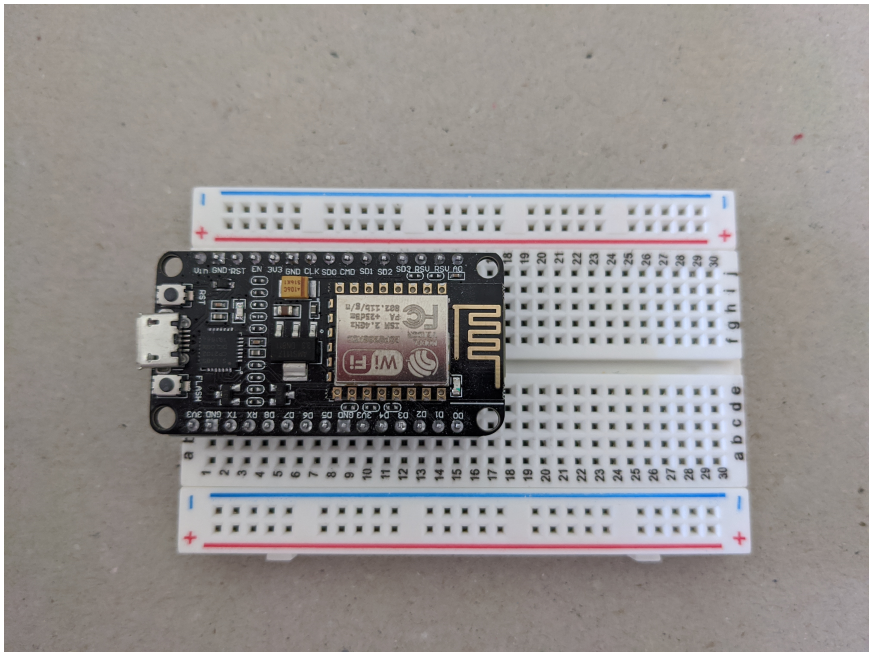


Figure 4.2: So far, so good!

Connect your microcontroller to the positive and negative rails of your breadboard

Using a jumper wire, pin one end of the wire into hole **A1** of the breadboard and the other end on *any* hole in the column labelled with a + on the side of the breadboard. This column is called the **positive rail**, and will eventually move power from your microcontroller around the circuit we are building.

Using another jumper wire, pin one end of the wire into hole **A2** and the other into *any* hole in the column labelled with a **-** on the side of the breadboard. This column is called the **negative rail** and, like the positive rail, helps to move power around the circuit we are building.

You should be left with something that looks like this:

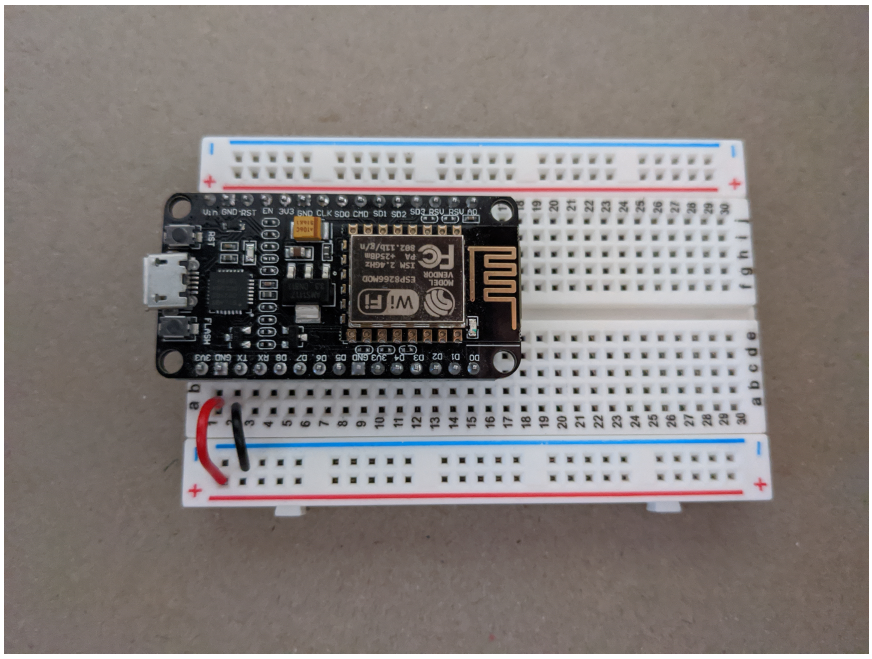


Figure 4.3: I'm absolutely POSITIVE I connected everything correctly!

Connect microcontroller to LED

Connect resistor between LED and negative power rail

4.2.2 Programming the microcontroller

4.3 Review

4.4 Possible Extensions

Chapter 5: Project 2: Button

Chapter 6: Project 3: LED Party

Chapter 7: Project 4: Sensor

7.1 Overview

Now that you have a handle on how to work with LEDs and buttons, let's look at something a little more complex. This project will introduce a humidity and temperature sensor. Over the course of this project, you will:

- Connect a sensor board to your microcontroller
- Import a library into your code that knows how to communicate with that sensor
- Use MicroPython to poll the sensor over and over to write out the current result
- Connect a screen to your microcontroller and output the sensor data to it

At the end of this project, your microcontroller should run a MicroPython program that prints out the current temperature and humidity of the room you're sitting in. Let's get started!

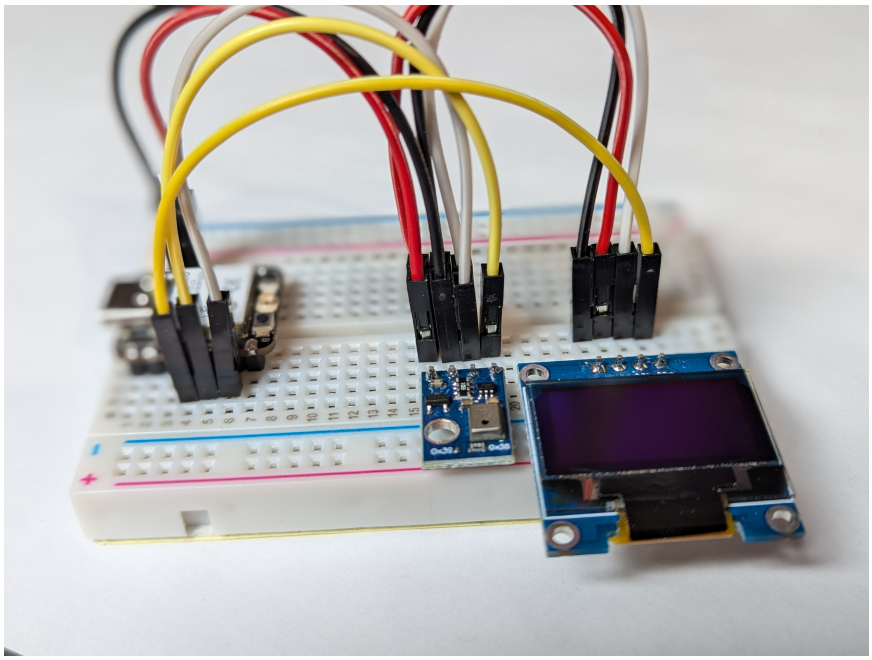


Figure 7.1: The end result should look something like this

7.2 Directions

7.2.1 Creating the circuit

Using jumper cables, you will be assembling a circuit between your microcontroller, your breadboard, and the small temperature/humidity board included in your kit.

Remove previous components

Before beginning, remove any components from prior chapters including LEDs, buttons, and wires. You may leave the microcontroller attached to the breadboard.

Attach the microcontroller to the breadboard

If it's not already, carefully insert the pins at the bottom of your microcontroller into the breadboard. Refer back to 1.2.2 for pin labels. When placing the board into the breadboard, make sure that the microcontroller is oriented such that:

- The pin labeled **5V** is inserted in hole at **Column H, Row 1** of the breadboard (or **H1**, for short)
- The pin labeled **GPIO2** is inserted in hole **D1** of the breadboard
- The pin labeled **GPIO20** is inserted in hole **H7** of the breadboard
- the pin labeled **GPIO21** is inserted in hole **D7** of the breadboard

You may need to apply more pressure than expected to seat the microcontroller properly in the breadboard. When its over, it should look like this:

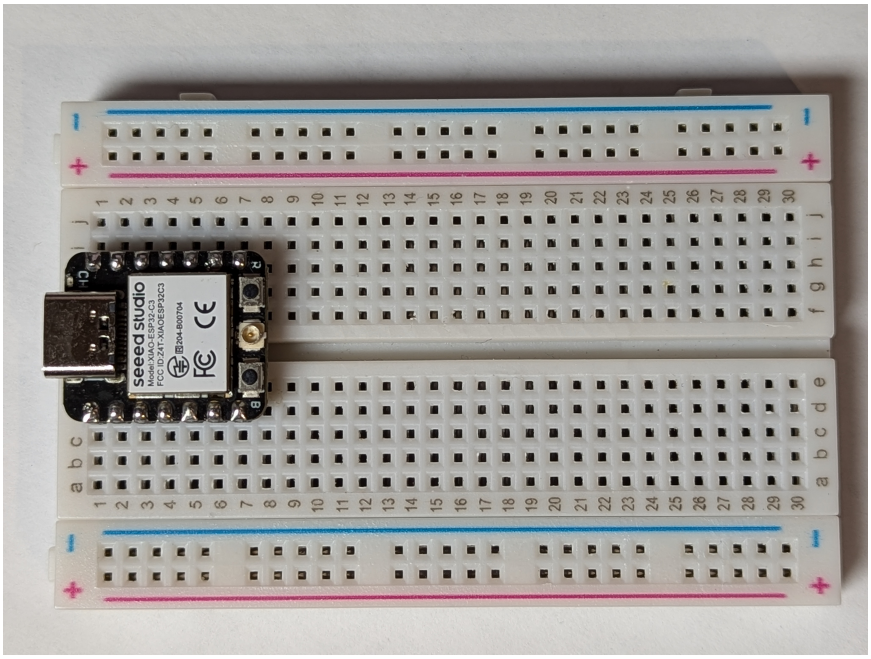


Figure 7.2: So far, so good!

Connect the necessary jumper wires

- Place a red jumper wire into hole **J3** of the breadboard and the other end in hole **D16** of the breadboard. This will provide **3.3** volts of power to the temperature/humidity sensor.
- Place a black jumper wire into hole **J2** of the breadboard and the other end in hole **D17** of the breadboard. This will provide the ground connection for the temperature/humidity sensor.
- Place a white jumper wire into hole **B6** of the breadboard and the other end in hole **D18** of the breadboard. This will provide a clock signal to the sensor board.
- Plase a yellow jumper wire into hole **B5** of the breadboard and the other end in hole **D19** of the breadboard. This will transmit data from the sensor back to the microcontroller.

You should be left with something that looks like this:

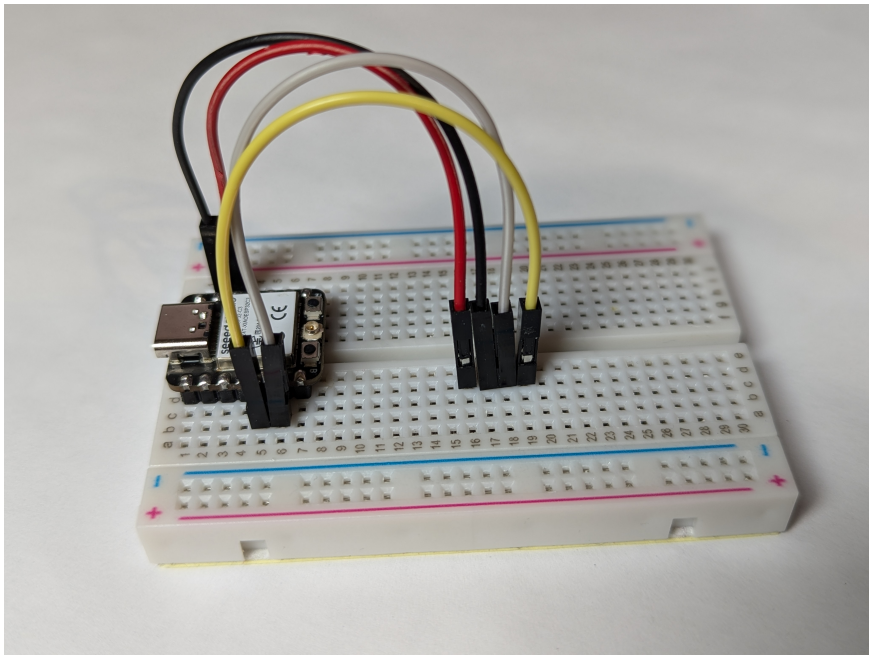


Figure 7.3: There are 4 wires that will be connected between the microcontroller and the sensor board.

Attach the temperature/humidity sensor to the breadboard

Plug the 4 pins of the sensor board into the breadboard just under where the 4 jumper wires are lined up. Make sure the pins of the board are lined up with the jumper wires and the board points away from them. It should look like this:

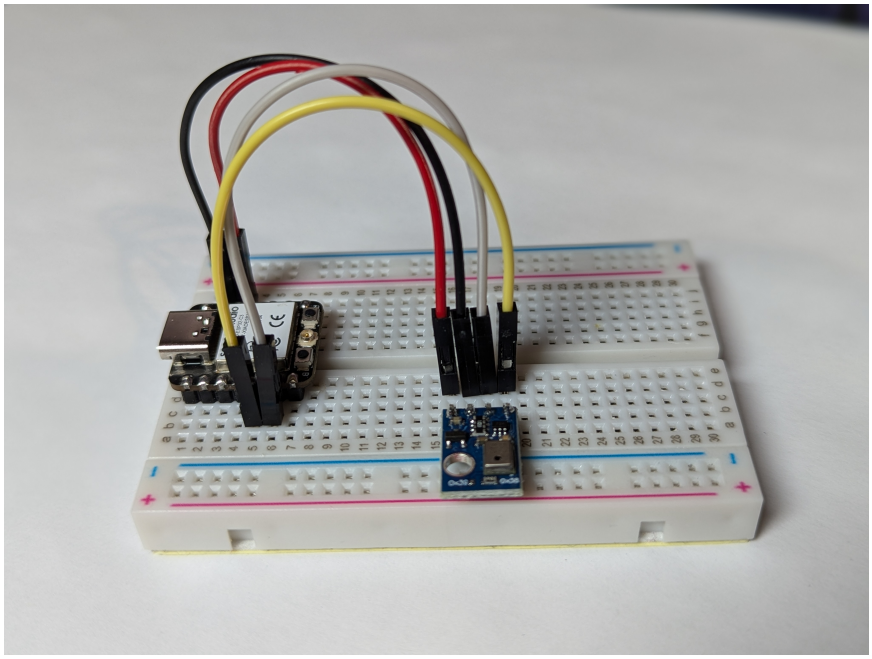


Figure 7.4: Click the button highlighted in red.

7.2.2 Programming the microcontroller

Once all of the wiring is correct, connect the USB cable to the microcontroller and load the IDE to access it. Refer back to Chapter 3 for instructions.

Click on the file named "project_4a_sensor.py". This will load the code in the editor for this section. Read through the comments and the code to get a sense for how it works. Once you are ready, you can click the blue play button in the upper left of the window to start the script:

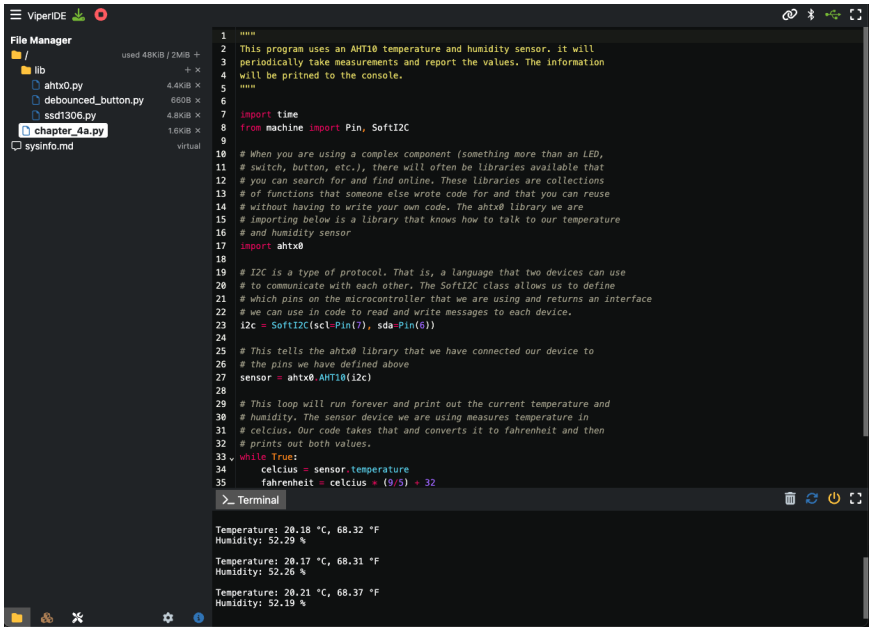


Figure 7.5: Once started, this script will run forever and output values into the terminal window. You can stop by pressing the red stop button.

7.2.3 Making it fancier

Having the temperature and humidity print out to the terminal is pretty cool (or pretty warm depending on where you are). But wouldn't it be better if we didn't have to be connected to a computer to see the values? We can give our device a screen and have it print the values out there as well.

Connect the necessary jumper wires

- First, disconnect the USB cable from your microcontroller. Doing this prevents accidentally connecting power to somewhere it shouln't go!
- Place a red jumper wire into hole **E16** of the breadboard and the other end in hole **E25** of the breadboard. This will provide **3.3** volts of power to the screen.
- Place a black jumper wire into hole **E17** of the breadboard and the other end in hole **E24** of the breadboard. This will provide the ground connection for the screen.
- Place a white jumper wire into hole **E18** of the breadboard and the other end in hole **E26** of the breadboard. This will provide a clock signal to the screen.
- Place a yellow jumper wire into hole **B4** of the breadboard and the other end in hole **E27** of the breadboard. This will transmit data from the microcontroller to the screen to be displayed.

You should be left with something that looks like this:

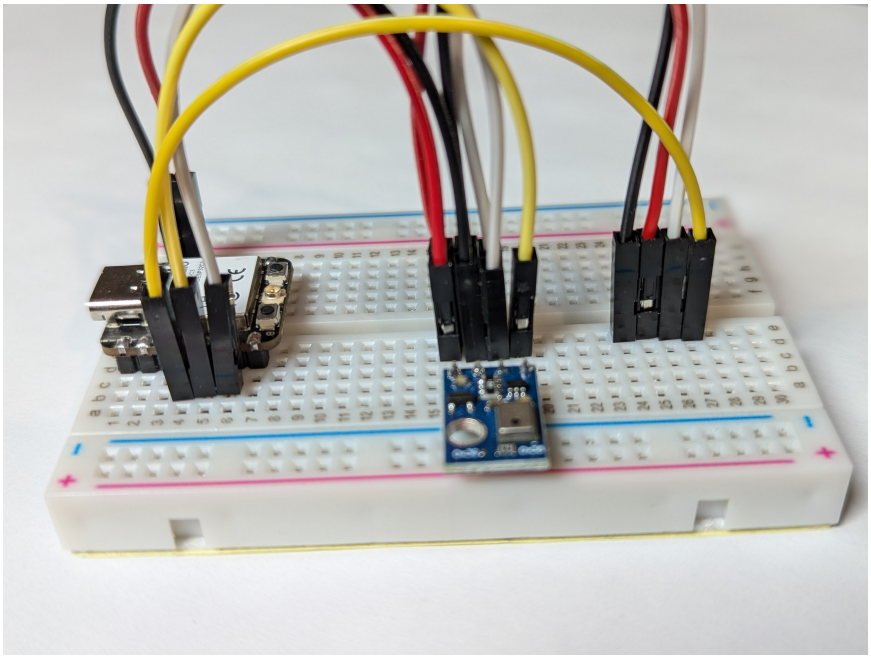


Figure 7.6: The first 3 wires are connected right behind the wires for the sensor. The last wire is connected to the microcontroller.

NOTE: The black and red jumper wires for the temperature sensor are in the reverse order as the black and red wires for the screen. If you get them backwards, your microcontroller will not be able to power on and it is possible to damage it if you leave it connected for too long.

Attach the screen to the breadboard

Plug the 4 pins of the screen into the breadboard just under where the 4 new jumper wires are lined up. Make sure the pins of the screen are lined up with the jumper wires and the screen points away from them. It should look like this:

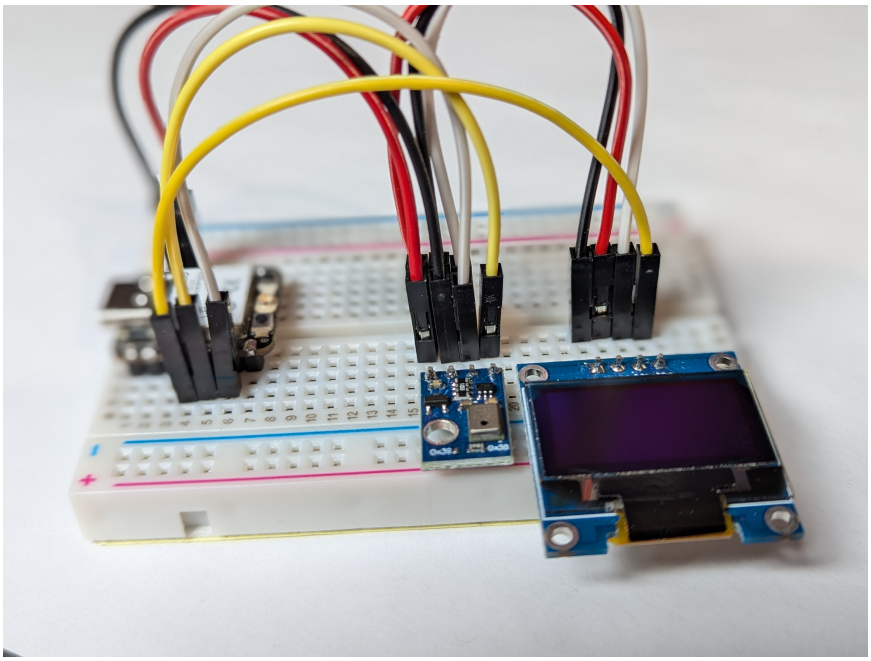


Figure 7.7: Click the button highlighted in red.

Plug the USB cable back into the microcontroller and reconnect to it on the IDE interface. Click on the file named "project_4b_sensor.py". This will load the code in the editor for this section. Read through the comments and the code to

get a sense for how it works. Once you are ready, you can click the blue play button in the upper left of the window to start the script:

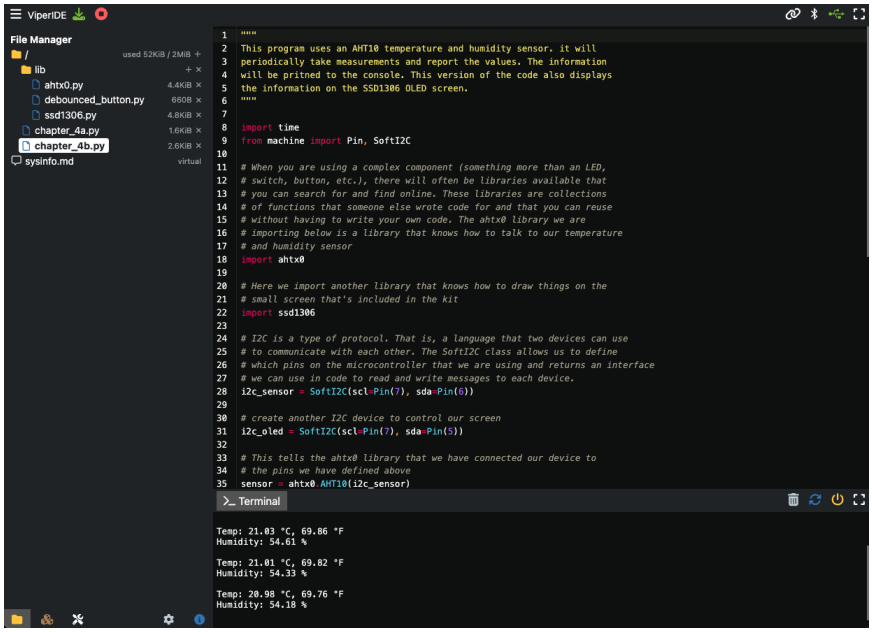


Figure 7.8: Once started, this script will run forever and output values into the terminal window and to the OLED screen. You can stop by pressing the red stop button.

7.3 Review

This project demonstrates how you can connect external sensors to a microcontroller, read their status, and output that status to a terminal or display. It also demonstrates how you can import external libraries for talking to these peripheral devices.

7.4 Possible Extensions

If you want to do some experimentation, try these:

- Connect an LED and have it light up when the temperature gets warmer than some threshold
- Connect a button and have the screen change between several different readouts when pressed

Chapter 8: Project 5: Game

8.1 Overview

If you've ever owned a GameBoy or similar device, then you might be familiar with what we're going to build as part of this project. This project will combine many of the skills from previous projects, so familiarity with those will help here. In this project you will:

- Connect buttons and a screen to your microcontroller
- Have a basic understanding of the code that implements a basic game loop including checking for input and drawing the current frame to the screen

At the end of this project, your microcontroller should run a MicroPython program that allows you to play a very basic version of Space Invaders. Let's get started!

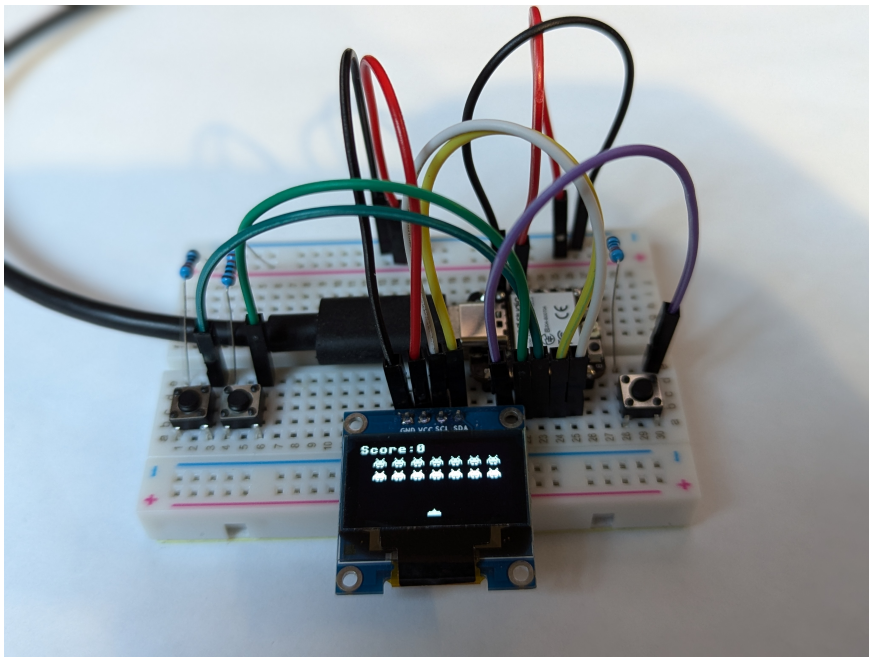


Figure 8.1: The end result should look something like this

8.2 Directions

8.2.1 Creating the circuit

Using jumper cables, you will be assembling a circuit between your microcontroller, your breadboard, and the buttons, resistors, and OLED screen included in your kit.

Remove previous components

Before beginning, remove all components from prior chapters including the microcontroller (as we will be putting it in a different location from the rest of the projects).

Attach the microcontroller to the breadboard

Carefully insert the pins at the bottom of your microcontroller into the breadboard. Refer back to 1.2.2 for pin labels. When placing the board into the breadboard, make sure that the microcontroller is oriented such that:

- The pin labeled **5V** is inserted in hole at **Column H, Row 20** of the breadboard (or **H20**, for short)
- The pin labeled **GPIO2** is inserted in hole **D20** of the breadboard
- The pin labeled **GPIO20** is inserted in hole **H26** of the breadboard
- The pin labeled **GPIO21** is inserted in hole **D26** of the breadboard

You may need to apply more pressure than expected to seat the microcontroller properly in the breadboard. When its over, it should look like this:

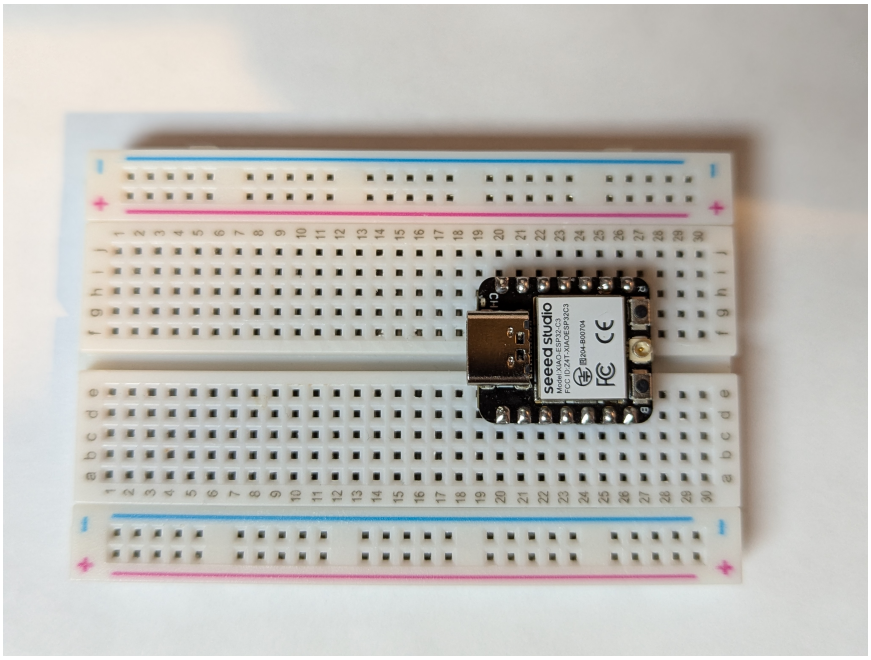


Figure 8.2: So far, so good!

Connect the buttons, resistors, and the screen

Place the buttons into the bottom corners of the breadboard. Two on the left, next to each other, and one on the right. Then connect a resistor for each button between the top positive bus (the red one) and the top left side of each button. Finally, place the OLED screen into the breadboard. Connect it's 4 pins into holes **A15**, **A16**, **A17**, and **A18**.

You should be left with something that looks like this:

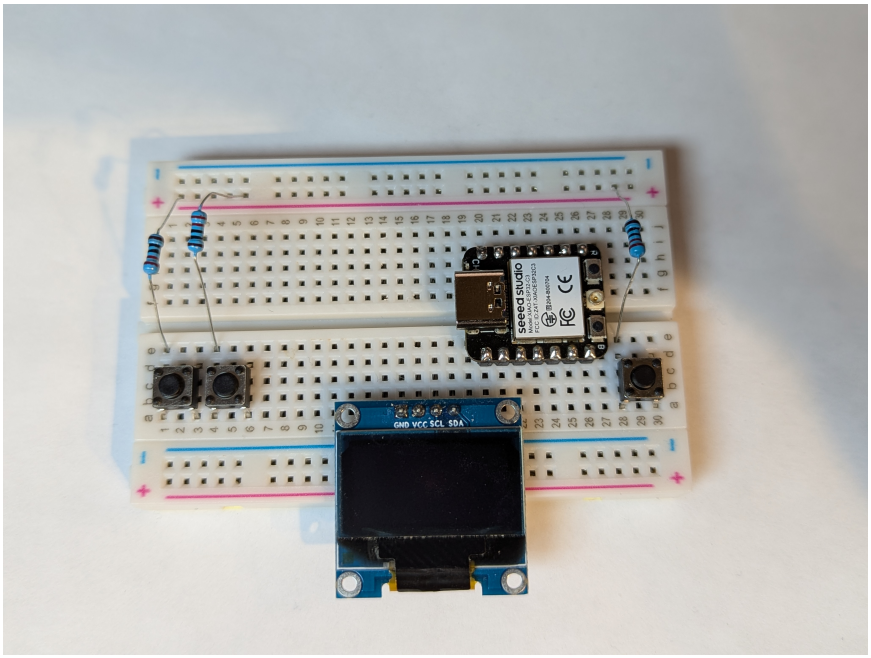


Figure 8.3: All of the components except for the jumper wires are now placed.

Connect the necessary jumper wires

- Place one end of a red jumper wire into hole **I22** of the breadboard and the other end into any hole on the top positive bus (the red one). This will provide **3.3** volts of power to all of the components.
- Using a black jumper wire, place one end of the wire into hole **I21** of the breadboard and the other end in any hole on the top negative bus (the blue one). This will provide the ground connection for all of the components.
- Place one end of a black jumper wire between the top negative bus and hole **B15**. This will provide the ground connection for the OLED screen.
- Place a red jumper wire between the top positive bus and hole **B16**. This will provide power for the OLED screen.
- Place a white jumper wire between **B17** and **B25**. This will provide a clock signal to the OLED screen.
- Place a yellow jumper wire between **B18** and **B24**. This will provide the data to display on the OLED screen.
- Place a green jumper wire between **E3** and **B23**. This will provide the signal for the left movement.
- Place a green jumper wire between **E6** and **B22**. This will provide the signal for the right movement.
- Finally, place a purple jumper wire between **E30** and **B21**. This will provide the fire signal.

You should be left with something that looks like this:

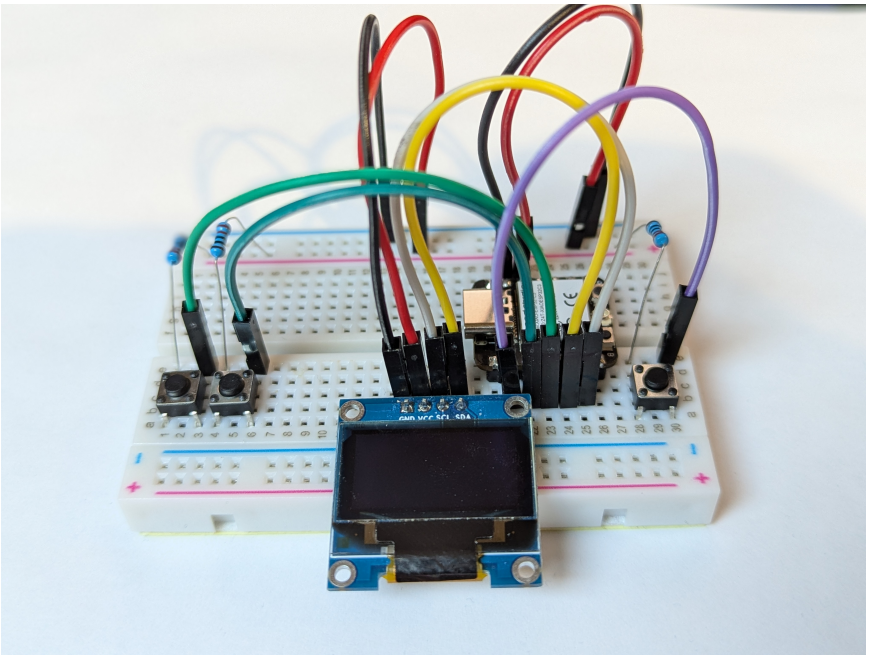


Figure 8.4: All components and wires are now installed

8.2.2 Programming the microcontroller

Once all of the wiring is correct, connect the USB cable to the microcontroller and load the IDE to access it. Refer back to Chapter 3 for instructions.

Click on the file named "project_5_game.py". This will load the code in the editor for this section. Read through the comments and the code to get a sense for how it works. Once you are ready, you can click the blue play button in the upper left of the window to start the game.

You can move your ship back and forth by pressing the buttons on the left. You can fire your ship's weapon by pressing the button on the right. Once you have destroyed all of the enemy ships, a "You Win!" screen will be shown and the game will freeze. If you want to start again, press the red stop button in the editor and then the blue play button again.

8.2.3 Examining the code

This project has the most code of any so far. The code can be broken down into several sections:

- The **Missle** class
- The **Enemy** class
- The **Ship** class
- The **game_loop()** function and a few helper functions (each called **update_<x>()**)
- A **main()** function at the bottom to get everything setup and started

Let's look at each of these below

The Missle class

Listing 8.1: The Missle class

```
1 class Missle:
2 """Keeps track of an on-screen missle"""
3
4 def __init__(self, x, y):
5     """Save our starting state"""
6     self.x = x
7     self.y = y
8     self.active = True
```

```
9
10  def move(self):
11      """If we move above the top of the screen, mark
12      ourselves as not active. The game loop should
13      cleanup any non-active missles.
14      """
15
16      global score, high_score
17
18      self.y -= 5
19      if self.y < 0:
20          self.active = False
21
22      # have we destroyed any enemies?
23      for enemy in enemies:
24          hitbox = enemy.hitbox
25          if not hitbox[0] <= self.x <= hitbox[0] + hitbox[2]:
26              continue
27          if not hitbox[1] <= self.y <= hitbox[1] + hitbox[3]:
28              continue
29          enemy.active = False
30          self.active = False
31          score += 10
32
33  def draw(self):
34      """Draw ourselves at our current posistion"""
35      for y_offset in range(5):
36          oled.pixel(self.x, self.y - y_offset, 1)
```

This class had 3 methods: an **__init__()** function that sets the initial state of the object, a **move()** function that is called for each frame the missle is on screen, and a **draw()** function which is called to tell the OLED how to display this missle.

In the **move()** function, the missle sets its position on the Y-axis upwards by 5 pixels. It then decides if it is now higher than the top of the screen and if so, marks itself as inactive. Then it checks if it has collided with any of the enemies that are still on the screen. If it has, it marks the enemy and itself as destroyed and increments the player's score by 10 points.

The **draw()** function is pretty simple. It just draws a vertical line by placing 5 pixels on top of each other, starting from the missle's current x and y coordinates.

The Enemy class

Listing 8.2: The Enemy class

```
1   class Enemy:
2   def __init__(self, x, y):
3       self.x = x
4       self.y = y
5       self.active = True
6       self.sprites = [space_invaders_sprites.ENEMY_SPRITE1, space_invaders_sprites.↘
            ENEMY_SPRITE2]
7       self.current_sprite = 0
8       self.sprite_changed = time.ticks_ms()
9
10  @property
11  def hitbox(self):
12      return (self.x, self.y, 16, 8)
13
14  def move(self):
15      pass
16
17  def draw(self):
18      now = time.ticks_ms()
```

```
19      if now - self.sprite_changed > 1000:
20          self.current_sprite += 1
21          self.current_sprite %= len(self.sprites)
22          self.sprite_changed = now
23      oled.blit(self.sprites[self.current_sprite], self.x, self.y)
```

This class had 3 methods and one special method called a property. The **__init__()** function sets the initial state of the object, the **move()** function that is called for each frame the enemy is on screen, the **draw()** function is called to tell the OLED how to display this enemy. There is also a **hitbox** property defined.

In the **__init__()** method, it sets some basic properties and then loads the sprite data from the external space_invaders_sprites.p file. Each sprite is just an array of data that describes which pixels should be drawn as white and which will be black. If you squint at it, you can see the shape of the enemy in the array.

The **hitbox** method is a special kind of method. It is decorated with **@property** which tells Python to call the method and return the computed value when the property is accessed. This allows the code to dynamically change the hitbox of the enemy if it is repositioned.

The **move()** method currently does nothing which means the enemies cannot move around the screen

The **draw()** method tells the OLED how to draw the enemy. It uses a function of the OLED library called **blit()** which takes in the sprite data that was defined earlier and draws it to the screen in one efficient operation. This is a much faster way that having to draw out each pixel in a loop.

The Ship class

Listing 8.3: The Ship class

```
1  class Ship:
2  """Keep track of the player's ship"""
3
4  def __init__(self, x, y):
5      """Save our initial state"""
6      self.x = x
7      self.y = y
8      self.last_fired = time.ticks_ms()
9
10 def move(self):
11     """Move left or right depending on which button is pressed"""
12
13     if left_button.value():
14         self.x -= 3
15     if right_button.value():
16         self.x += 3
17     if self.x > 119:
18         self.x = 119
19     if self.x < 9:
20         self.x = 9
21
22 def draw(self):
23     """Draw the pixels of our ship"""
24
25     oled.blit(space_invaders_sprites.SHIP_SPRITE, self.x - 8, self.y - 5)
26
27 def fire(self):
28     """If the fire button is pressed and the cooldown time has
29     passed, then generate a new missle object at our current position
30     """
31
32     now = time.ticks_ms()
33     if fire_button.value() and now - self.last_fired > 200:
34         self.last_fired = now
35         missles.append(Missle(self.x, self.y - 10))
```

This class had 4 methods, the **__init__()** function sets the initial state of the object, the **move()** function that is called for each frame the ship is on screen, the **draw()** function is called to tell the OLED how to display the ship, and the **fire()** function which checks if the fire button is pressed and creates a missle at the current ship position.

In the **move()** method, it checks if either of the left or right movement buttons are currently pressed. If so, it moves the ship in that direction. If it detects that this ship is currently at the edges of the screen, it doesn't move it any further.

In the **draw()** method, we use the **blit()** method mentioned previously to draw our ship's sprite to the OLED.

In the **fire()** method, we chekc if the fire button is currently held down. If so, it creates a new instance of the **Missle** class and sets it's initial position to be just above the ship's current position.

The game loop and helper function

Listing 8.4: The game loop and helper functions

```
1  def update_missles():
2      """Remove any non-active missles and move the rest"""
3
4      global missles
5      missles = [m for m in missles if m.active]
6      for missle in missles:
7          missle.move()
8          missle.draw()
9
10 def update_enemies():
11     """Remove any destroyed enemies and move the rest"""
12
13     global enemies
14     enemies = [e for e in enemies if e.active]
15     for enemy in enemies:
16         enemy.move()
17         enemy.draw()
18
19 def update_score():
20     global game_on
21     oled.text("Score:%s" % score, 0, 0)
22     if not enemies:
23         oled.text("You Win!", 35, 30)
24         game_on = False
25
26 def game_loop(ship):
27     """Drive the main game loop"""
28
29     oled.fill(0)
30     ship.fire()
31     update_missles()
32     update_enemies()
33     ship.move()
34     ship.draw()
35     update_score()
36     oled.show()
```

The **game_loop()** function is run over and over while the game is ongoing. It is responsible for the order that all game objects are checked and redrawn. At the start of the loop it clears the entire screen (fills all of the pixels with black). Then it checks if any new missles need to be created, updates the existing missles, updates the enemies and the ship, and displays the current score (which also checks if the game is over). After all objects have been updated, it tells the OLED to display everything that's been written to it for this frame.

The main function

Listing 8.5: The main function

```
1      def main():
```

```
    try:
        ship = Ship(64, 64)
        for y in range(12, 36, 12):
            for x in range(10, 118, 16):
                enemies.append(Enemy(x, y))
        while game_on:
            game_loop(ship)
    except KeyboardInterrupt:
        pass

main()
```

The **main()** function is pretty short, but very important. It creates instances of our ship and all of the enemies and then runs the **game_loop()** function over and over until the game is complete.

8.3 Review

8.4 Possible Extensions

If you want to do some experimentation, try these:

- Add some LEDs and have them light up when an enemy is hit or when the game is won.
- Have the enemies move around the screen. In the classic space invaders game, they moved down towards the ship as the game progressed.

Chapter 9: Project 6: Web

9.1 Overview

A piece of hardware that we haven't talked about yet which is built into the ESP32-C3 microcontroller is it's WiFi processor. We can make use of this to connect to the internet and pull in all kinds of data or transmit data to other services! If you've ever heard the phrase "Internet of Things", this is what they mean. Over the course of this project, you will:

- Use the embedded WiFi processor to connect to the internet
- Make requests to free 3rd party API services to fetch live data
- Use the connected buttons and screen to display different sets of that data

At the end of this project, your microcontroller should run a MicroPython program that displays 3 different screens of data that you can rotate through with the buttons. Let's get started!

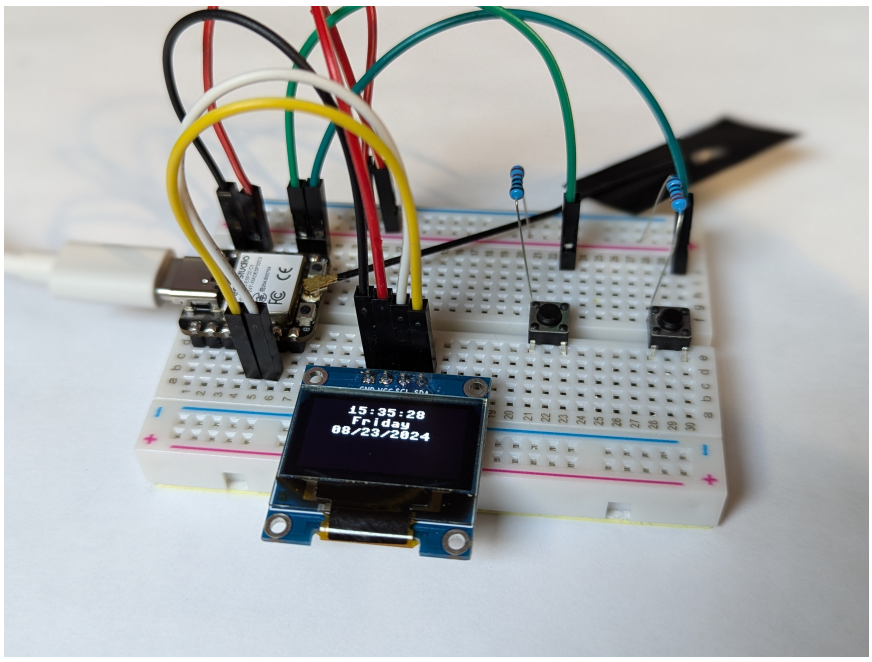


Figure 9.1: The end result should look something like this

9.2 Directions

9.2.1 Creating the circuit

Using jumper cables, you will be assembling a circuit between your microcontroller, your breadboard, two buttons and the small OLED screen included in your kit.

Remove previous components

Before beginning, remove any components from prior chapters including LEDs, buttons, and wires. You may leave the microcontroller attached to the breadboard.

Attach the microcontroller to the breadboard

If it's not already, carefully insert the pins at the bottom of your microcontroller into the breadboard. Refer back to 1.2.2 for pin labels. When placing the board into the breadboard, make sure that the microcontroller is oriented such that:

- The pin labeled **5V** is inserted in hole at **Column H, Row 1** of the breadboard (or **H1**, for short)
- The pin labeled **GPIO2** is inserted in hole **D1** of the breadboard
- The pin labeled **GPIO20** is inserted in hole **H7** of the breadboard
- the pin labeled **GPIO21** is inserted in hole **D7** of the breadboard

You may need to apply more pressure than expected to seat the microcontroller properly in the breadboard. When its over, it should look like this:

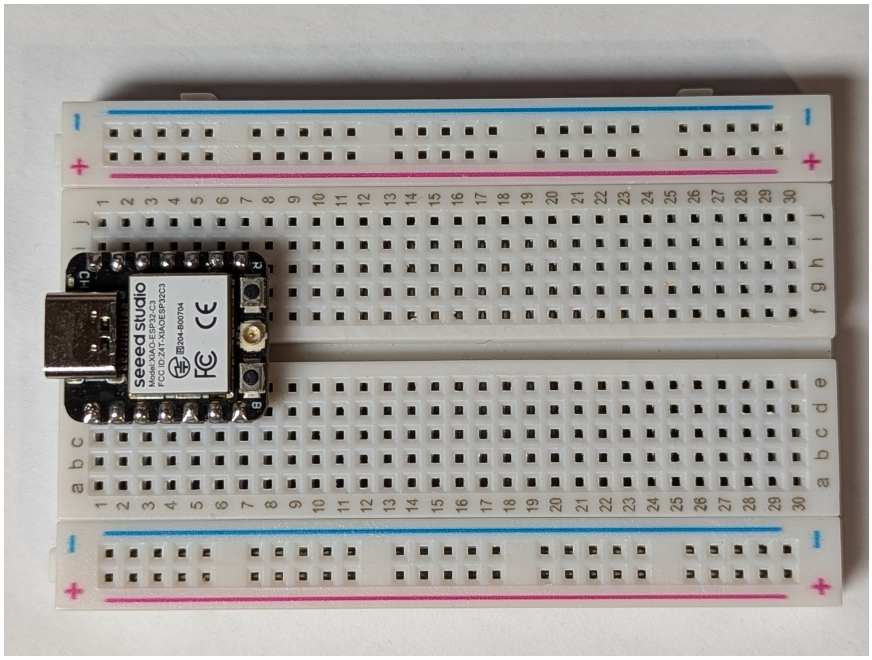


Figure 9.2: So far, so good!

Place the OLED Screen and Buttons on the Board

Place the buttons towards the right side of the breadboard. Then connect a resistor for each button between the top positive bus (the red one) and the top left side of each button. Finally, place the OLED screen into the breadboard. Connect it's 4 pins into holes **A12**, **A13**, **A14**, and **A15**.

You should be left with something that looks like this:

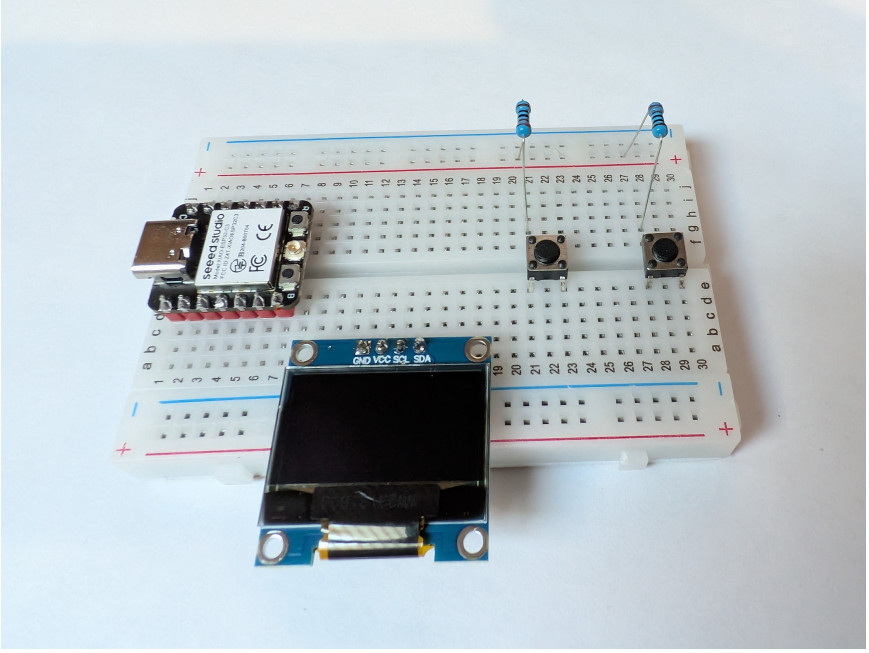


Figure 9.3: All of the components except for the jumper wires are now placed.

Connect the necessary jumper wires

- Place one end of a red jumper wire into hole **I2** of the breadboard and the other end into any hole on the top positive bus (the red one). This will provide **3.3** volts of power to all of the components.
- Using a black jumper wire, place one end of the wire into hole **I2** of the breadboard and the other end into **C12**, which is the pin marked as GND on the OLED.
- Place a red jumper wire between the top positive bus and hole **C13**. This will provide power for the OLED screen.
- Place a white jumper wire between **C14** and **B25**. This will provide a clock signal to the OLED screen.
- Place a yellow jumper wire between **C15** and **B24**. This will provide the data to display on the OLED screen.
- Place a green jumper wire between **J23** and **J6**. This will provide the signal to move to the previous screen.
- Place a green jumper wire between **J30** and **J7**. This will provide the signal to move to the next screen.
- Finally, attach the antenna to the microcontroller by plugging the gold end of its wire into the socket in the bottom of the microcontroller's board, between the buttons.

You should be left with something that looks like this:

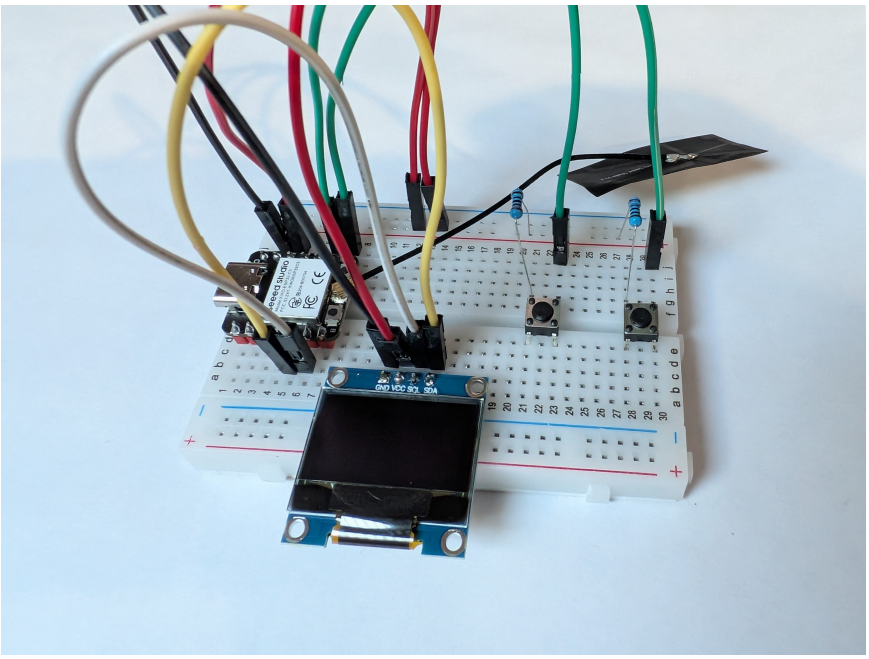


Figure 9.4: All components and wires are now installed

9.2.2 Programming the microcontroller

Once all of the wiring is correct, connect the USB cable to the microcontroller and load the IDE to access it. Refer back to Chapter 3 for instructions.

Click on the file named "project_6_web.py". This will load the code in the editor for this section. Read through the comments and the code to get a sense for how it works. You will need to edit the code and provide values for the WiFi network name and password before it will work:

```
# You must provide values for these that match the WiFi network you would like to
# connect to.
NETWORK_NAME = ""
NETWORK_PASSWORD = ""
```

Figure 9.5: If you are doing this during the workshop at NetApp, these will be given to you. If you are at home, use the credentials for your house.

Once you are ready, you can click the blue play button in the upper left of the window to start the program.

You can press the left and right buttons to rotate through the different views on the screen. Note that it takes a few seconds for the microcontroller to load and parse the data from each API. This can sometimes lead to buggy behavior when pressing the buttons quickly.

9.2.3 Examining the code

This project's code shows a few new concepts:

- Configuring the microcontroller's WiFi hardware and connecting to a network
- Making requests to a web-based API
- Parsing the results of the API and displaying them on our screen

Let's take each of these and examine them closer

Connecting to WiFi

Listing 9.1: WiFi Code

```
1 import network
```

```
2
3  # You must provide values for these that match the WiFi network you would like to
4  # connect to.
5  NETWORK_NAME = ""
6  NETWORK_PASSWORD = ""
7
8  def connect_to_wifi():
9      if not NETWORK_NAME or not NETWORK_PASSWORD:
10         print("Don't forget to fill in the WiFi details at the top of the script!")
11         return False
12
13     print(f"Connecting to {NETWORK_NAME}...")
14     OLED.text("Connecting...", 12, 25)
15     OLED.show()
16
17     interface = network.WLAN(network.STA_IF)
18     interface.active(True)
19     interface.connect(NETWORK_NAME, NETWORK_PASSWORD)
20     tries = 10
21     while tries > 0:
22         if not interface.isconnected():
23             tries -= 1
24             time.sleep(1)
25         else:
26             print(f"Connected to {NETWORK_NAME} successfully")
27             break
28     else:
29         print(f"Failed to connect to {NETWORK_NAME}. Is the password correct?")
30         return False
31
32     return True
```

In the listing above, we see just the code that is responsible for setting up the WiFi hardware and connecting to the network. First we import the network module. This is part of the standard library for MicroPython. In the **connect_to_wifi()** function, we make sure the network name and password are set. Then we print out a status message to let the user know that something is happening. Then we get the **interface** object by asking the **network** library to give us a handle to the microcontroller's WiFi chip. We mark the **interface** as active and tell it to connect with the credentials the user gave us.

It takes a little time to connect to the WiFi router, so we have a loop to wait until that happens. If we get connected, then we will exit the function returning **True**. If something prevents us from conencting, for example a mistyped password, then we will print a message to let the user know and return **False**.

Talking to an API

Listing 9.2: Making Web Requests

```
1  def draw_date_and_time():
2      # Use the timeapi.io site to fetch the current time
3      # using the IP address of our microcontroller
4      # https://timeapi.io/swagger/index.html
5      response = requests.get(f"https://timeapi.io/api/time/current/ip?ipAddress={↘
           IP_ADDRESS}")
6      date_time = response.json()
7      OLED.fill(0)
8      OLED.text(f"{date_time['hour']:02d}:{date_time['minute']:02d}:{date_time['seconds↘
           ']:02d}", 30, 5)
9      OLED.text(f"{date_time['dayOfWeek']}", 35, 15)
10     OLED.text(f"{date_time['date']}", 20, 25)
```

In the above code, we make use of our new network connection by making an HTTP GET request to the https://timeapi.io/ site. This site provides a JSON-based RESTful API that anyone can use. The URL that we make the request to is asking for the current time at the IP address that we were assigned when we connected to WiFi. If we made this request in our browser, the response would look something like this:

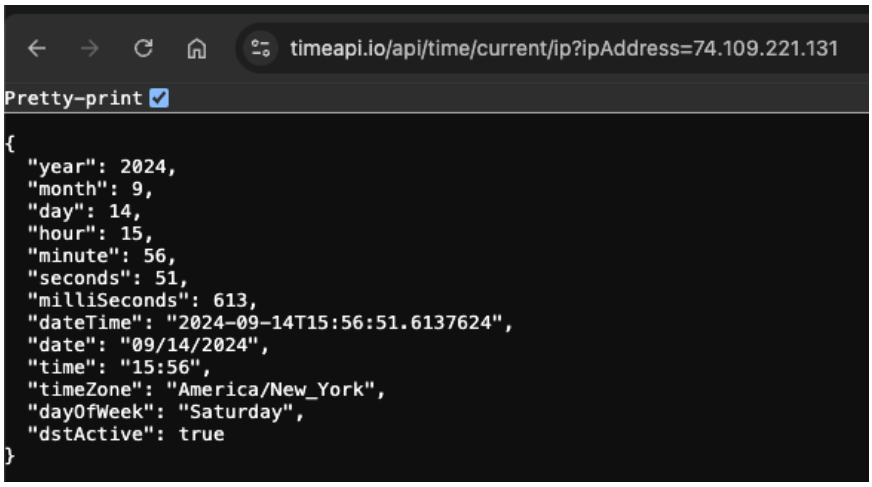


Figure 9.6: An example response from the timeapi.io site

This format is called JSON. It is a way of expressing groups of data that many programming languages can understand. It is one of the most common formats for web-based APIs. In our code, we use the **response.json()** function to parse this response into a Python dictionary object. We can then access the fields of the response in our code and print them out on our screen.

9.3 Review

This chapter is an example of how you might create an Internet of Things device using a microcontroller and some Python code. While our example is small, you could imagine that instead of our small screen, we are displaying this on a large touch screen, perhaps one acting as the bathroom mirror. Then when you were getting ready each day, the things most relevant to you would be readily visible.

9.4 Possible Extensions

If you want to do some experimentation, try these:

- Change the code to auto-cycle between the screens every X seconds even without a button press
- Find another web-based API that is free and accessible and display information from it on a 4th screen

Chapter 10: Project 7: Sound

10.1 Overview

Included in your kit is a small speaker. The process of making sound with any speaker, including this one, is to send it a signal as a series of voltages. A change in voltage engergizes or de-energizes a coil of wire in the speaker which attracts or repels the magnet it contains. This moves the diaphram of the speaker which vibrates the air and makes sound. We can make all of this happen using our microcontroller and a little bit of code to controll the pin output very quickly. Over the course of this project, you will:

- Set up a pin of the microcontroller to act as a speaker controller
- Write code to interpret a list of notes into pin on/off signals
- Learn about Pulse Width Modulation and how it can affect the sound produced

At the end of this project, your microcontroller will be set up to play sounds out of the attached speaker. Let's get started!

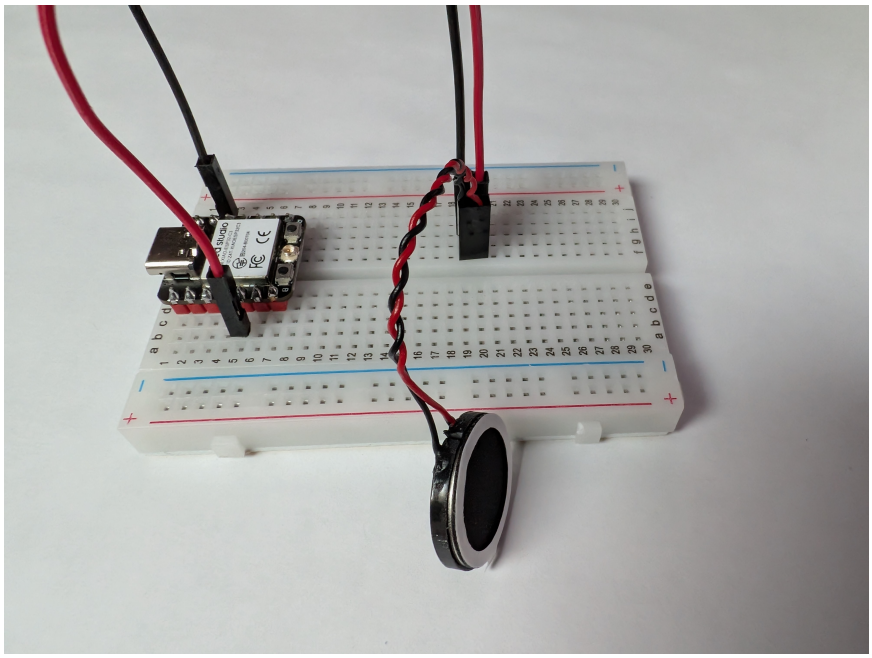


Figure 10.1: The end result should look something like this

10.2 Directions

10.2.1 Creating the circuit

Using jumper cables, you will be assembling a circuit between your microcontroller, your breadboard, and the speaker included in your kit.

Remove previous components

Before beginning, remove any components from prior chapters including LEDs, buttons, and wires. You may leave the microcontroller attached to the breadboard.

Attach the microcontroller to the breadboard

If it's not already, carefully insert the pins at the bottom of your microcontroller into the breadboard. Refer back to 1.2.2 for pin labels. When placing the board into the breadboard, make sure that the microcontroller is oriented such that:

- The pin labeled **5V** is inserted in hole at **Column H, Row 1** of the breadboard (or **H1**, for short)
- The pin labeled **GPIO2** is inserted in hole **D1** of the breadboard
- The pin labeled **GPIO20** is inserted in hole **H7** of the breadboard
- the pin labeled **GPIO21** is inserted in hole **D7** of the breadboard

You may need to apply more pressure than expected to seat the microcontroller properly in the breadboard. When its over, it should look like this:

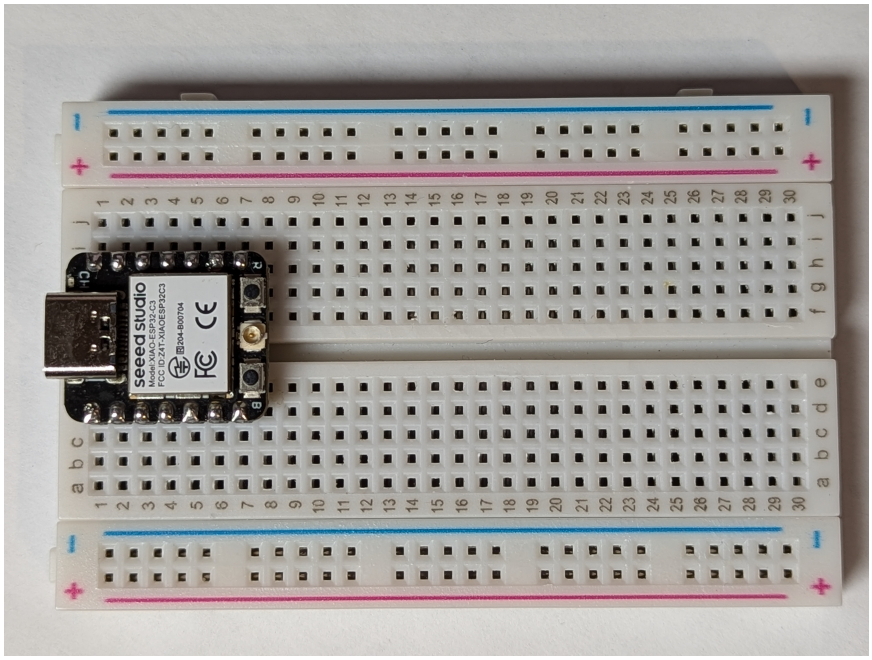


Figure 10.2: So far, so good!

Place the Speaker on the Board

Place the speaker into the board such that the black wire is in hole **F19** and the red wire is in hole **F20**.

You should be left with something that looks like this:

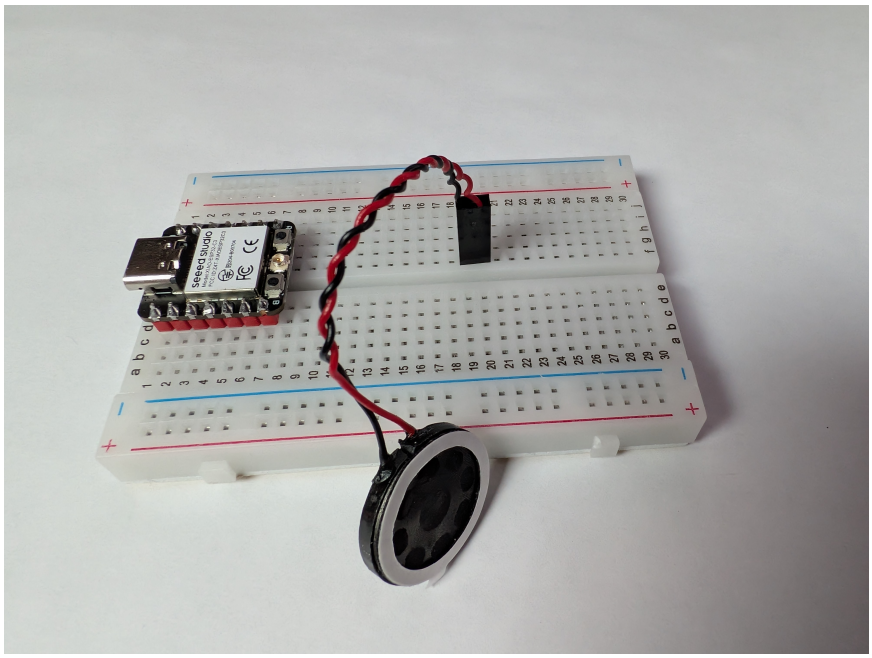


Figure 10.3: All of the components except for the jumper wires are now placed.

Connect the necessary jumper wires

- Place one end of a red jumper wire into hole **B5** of the breadboard and the other end into **H20**. This will provide the on/off signal to the speaker according to our program's needs.
- Using a black jumper wire, place one end of the wire into hole **I2** of the breadboard and the other end into **C12**, matching the black wire of the speaker. This will provide it's ground connection.

You should be left with something that looks like this:

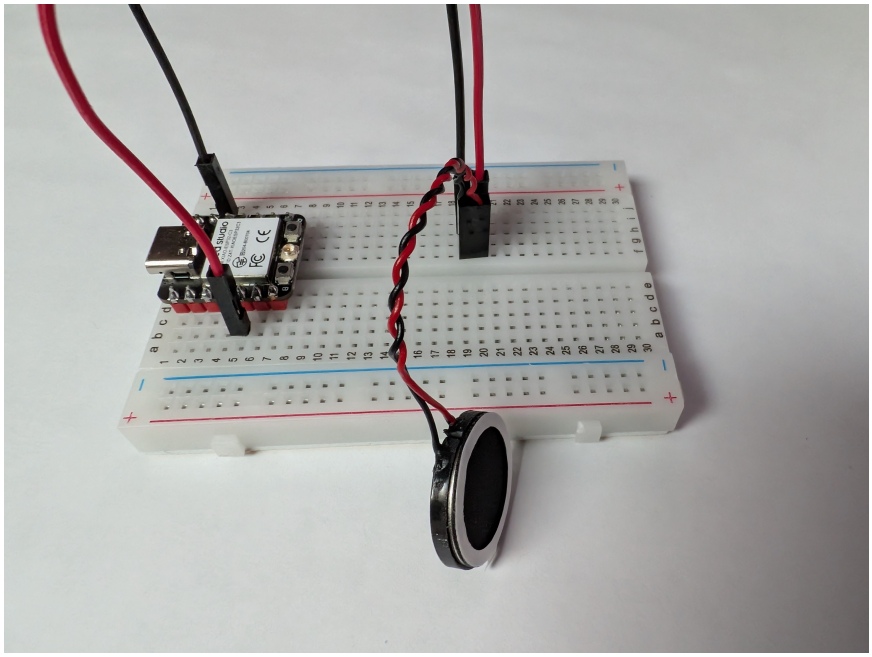


Figure 10.4: All components and wires are now installed

10.2.2 Programming the microcontroller

Once all of the wiring is correct, connect the USB cable to the microcontroller and load the IDE to access it. Refer back to Chapter 3 for instructions.

Click on the file named "project_7_sound.py". This will load the code in the editor for this section. Read through the comments and the code to get a sense for how it works. Once you are ready, you can click the blue play button in the upper left of the window to start the program.

The speaker should start playing sound. See if you can recognize the tune!

10.2.3 Examining the code

This project's code shows a few new concepts:

- Configuring a pin on the microcontroller as a PWM device
- Using **time.sleep_ms()** calls in our code to wait a certain amount of time

Let's take each of these and examine them closer

Setting up PWM

Listing 10.1: PWM Mode

```
1       # set the notes and sleep for the appropriate time
2       pwms = [PWM(pin, freq=note, duty=instrument) for pin in pins]
```

PWM stands for Pulse Width Modulation. When we configure the PWM object by passing a pin object, it instructs the microcontroller to start pulsing that pin at the given frequency and duty cycle. By pulsing, we mean that the voltage on the pin goes up and down at a fixed rate. If you've heard that humans can hear frequencies between 20Hz and 20,000Hz, this is an example of that same thing. In this case, we are going to pulse the pin at a certain frequency based on the note we want to hear.

The code has a list of notes and the associated frequencies in it. Here's a snippet:

Listing 10.2: Note names mapped to frequencies

```
1       ("G#0", 208),
2       ("A1", 220),
3       ("Bb1", 233),
4       ("B1", 247),
5       ("C1", 262), # middle C
6       ("C#1", 277),
7       ("D1", 294),
```

From this list, we can see that middle C (that is, the note named C that is in the middle of a full sized piano) is associated with a frequency of 262Hz. Or in other words, if we want to hear a middle C out of our speaker, then we need to pulse the pin on and off 262 times per second.

The other important argument is releated to the duty cycle. Duty cycle means the amount of time the pin is on vs. the amount of time the pin is off. For example, a 50% duty cycle is when the pin is on for half of the time and off for half of the time. A 75% duty cycle is when the pin is on for 3/4 of the time and off for 1/4 of the time. Here's what that would look like visually:

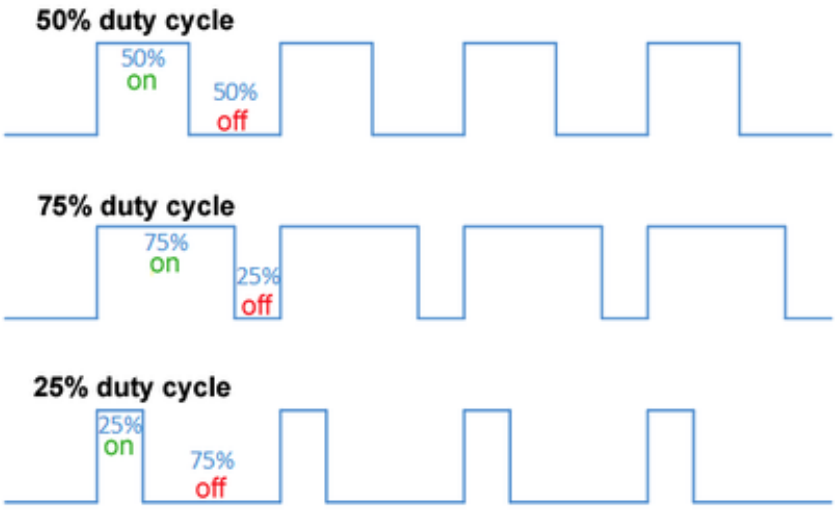


Figure 10.5: Examples of 3 different duty cycle settings

You can play with the duty cycle to make the resulting notes sound like different instruments. In the code, this is done with this mapping:

Listing 10.3: Instrument names mapped to duty cycle settings

```
INSTRUMENTS = {
    "trumpet": 64,
    "clarinet": 512,
    "keyboard": 700,
}
```

The numbers here represent the number of milliseconds (out of 1000 milliseconds in 1 second) that the pin should be on. The instrument names give some idea about how it might sound, but these are only very rough approximations!

Going to sleep

Another important part of music (and some might say the most important part) is resting. A rest in music is a period of time when no note is being played. In order to implement this in our code, we call the **time.sleep_ms()** function and then return without setting any of the pins to pulse. The **time.sleep_ms()** tells the microcontroller to do nothing for the given amount of time before it moves to the next line of code.

Listing 10.4: Using the sleep function

```
    # if this is a rest, then just sleep
    if not note:
        time.sleep_ms(duration)
        return
```

10.3 Review

In this project, you saw how a microcontroller plus some code can be used to make a speaker make sounds. While this project only makes simple types of sounds, you can get much fancier with some additional code and hardware and play .mp3 files from an attached SD card or even stream web-based radio stations!

10.4 Possible Extensions

If you want to do some experimentation, try these:

- Try playing with the duty cycle to see if you can come up with other instrument sounds
- Try creating your own songs by using the same format as the example song

Appendix A: Python Primer

If you're new to Python, this section will give you a few things you should know in order to better understand the projects in this guide. This is by no means a complete or comprehensive look at the Python language. For that, we recommend looking at the official Python site and reading through the tutorial there.

> Note: for the projects being used here, we are using an implementation of Python known as MicroPython. This version is meant to run on microcontrollers with limited resources. It also has built into it libraries for dealing with hardware devices that are not part of the standard CPython distribution. Therefore, not all Python examples you find online will run on your microcontroller and not all projects for a microcontroller can be run on your computer. But a lot of the code can be shared so the lessons you learn here can apply to other Python projects.

Here is a sample of a small Python script. We will disect and explain what each section does below:

Listing A.1: An example Python script

```
1  def show(message, repeat=1):
2      """This function prints the given message to the
3      console as smany times as specified in the
4      srepeat parameter.
5      """
6
7      for iteration in range(0, repeat):
8          print(iteration, message)
9
10 name = input("What is your name: ")
11 show(name)
12 show(name, repeat=3)
```

On line 1, we are defining a function named show. This function accepts two parameters, message and repeat. The message parameter is required and the repeat parameter is optional with a default value of 1.

Lines 2 through 5 comprise the docstring for the function. This information is meant for programmers to read and explains what the function does. It does not affect how the function works.

Line 7 starts a loop. The loop will repeat the statements in the loop body until a condition is met. In this case, it will loop until it has performed the operation for each repeat.

Line 8 is the body of the loop. This statement will print the message that the user passed in to the console along with the iteration number of the loop.

Line 10 prompts the user for their name and saves the result in a variable called name.

Line 11 calls our show function which will print the user's name once (the default).

Line 12 calls our show function again, this time saying that we want to repeat the loop of printing the name twice.

Running the program, we will see output like this:

```
$ python program.py
What is your name: Emily
0 Emily
0 Emily
1 Emily
2 Emily
$
```

Another feature that you'll see used often in Python are classes. Classes are a convienent way to model something in your program that holds state and implements functionality. For example, let's say that we are writing a game about racing go- karts. We need to allow each player to have their own kart and keep track of how fast it is going, which way they are turning, and allow the kart to speed up and slow down. Here is a small class that will help us do that:

Listing A.2: An example of a Python class

```
1  class Kart:
2      MAXIMUM_SPEED = 100
```

```
3
4       def __init__(self):
5           """The kart starts motionless at the beginning"""
6           self._speed = 0
7           self._direction = 0
8           self._acceleration = 0
9
10      def brake(self):
11          """This is called when the user presses the brake button"""
12          self._acceleration = -5
13
14      def accelerate(self):
15          """This is called when the user presses the accelerator button"""
16          self._acceleration = 5
17
18      def steer(self, direction):
19          """This is called when the user presses left or right"""
20          self._direction = direction
21
22      def update(self, ticks):
23          """Update will be called by our game engine and will be
24          provided the number of ticks since it was last called.
25          """
26
27          self._speed += self._acceleration * ticks
28
29          # limit our speed so that we don't go faster than our
30          # kart is allowed to, or slower than 0
31          if self._speed > Kart.MAXIMUM_SPEED:
32              self._speed = Kart.MAXIMUM_SPEED
33          if self._speed < 0:
34              self._speed = 0
```

Looking at this class, there are 5 methods. The first one (on line 4) is a special method that is called by the Python interpreter whenever a new Kart is created. It will initialize some variables for this particular Kart object.

You may have noticed that the first method takes a parameter called "self". This is the first parameter of all methods in a class in Python. It is automatically passed by the interpreter and is a reference to the current object. It lets us access the variables that belong to the class, like those we defined in the __init__ method.

Speaking of the variables in the __init__ method. Notice how we named them all with an underscore? This is a convention in Python that says they are private to our class and that code written outside of the class shouldn't access them directly. That means that our class should provide ways to modify or read these variables via other methods.

The second method starts on line 10. This is called when the player presses the brake button on their controller and will set our Kart's acceleration to a negative value so that we start to slow down. It modifies the private _acceleration member of the class.

The third starts on line 14. It is the opposite of braking and will start speeding our Kart up when the user presses the accelerator. It aslo modifies the private _acceleration member of the class.

The fourth method, line 18, is again something to deal with user input. This time we can see that it takes a second parameter, direction. If the user presses left on their controller, then we can expect left to be passed here. The same for right. We will modify the private _direction member here.

Finally, we have a fifth method starting on line 22. This method is called by our game engine and uses the class members to determine what happens to the Kart throughout the game. That is, it is asking the Kart to update itself at a certain moment in time (usually once per frame) so that next time it draws it to the screen, it will be in the updated location.

Notice in the last method, we are accessing not only our own variables, _speed, and _acceleration, but we are also reading a class variable, Kart.MAXIMUM_SPEED. Unlike our member variables, a class variable is the same for all instances of a class. It is useful here to keep the game fair so that all Karts have the same limitation on their speed.