

MicroPython and Microcontrollers

NetApp YWIT

September 26, 2024

Contents

1	Introduction	2
1.1	What this course will teach	2
1.2	Working with the microcontroller	2
1.2.1	Installing the driver	2
1.2.2	Pinout Reference	2
2	Project 1: Blink	4
2.1	Overview	4
2.2	Directions	4
2.2.1	Creating the circuit	4
2.2.2	Programming the microcontroller	5
2.3	Review	5
2.4	Possible Extensions	5
3	Project 2: Button	7
4	Project 3: LED Party	8
5	Project 4: Sensor	9
5.1	Overview	9
5.2	Directions	10
5.2.1	Creating the circuit	10
5.2.2	Programming the microcontroller	11
5.2.3	Making it fancier	13
5.3	Review	15
5.4	Possible Extensions	15
6	Project 5: Game	16
6.1	Overview	16
6.2	Directions	17
6.2.1	Creating the circuit	17
6.2.2	Programming the microcontroller	19
6.2.3	Examining the code	21
6.3	Review	24
6.4	Possible Extensions	24
7	Project 6: Sound	25
8	Project 7: Chat	26
A	Electronics Essentials	27
B	Python Primer	28

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 building several projects ranging from simple to more complex. These projects are based on the ESP32-C3 microcontroller 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, 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: The 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

Project 1: Blink

2.1 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!

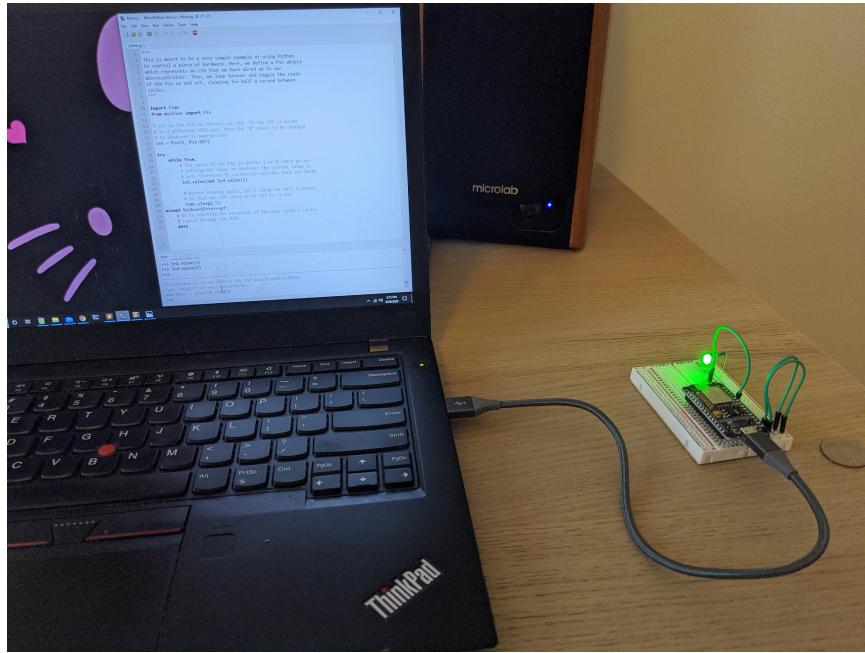


Figure 2.1: The end result should look something like this

2.2 Directions

2.2.1 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:

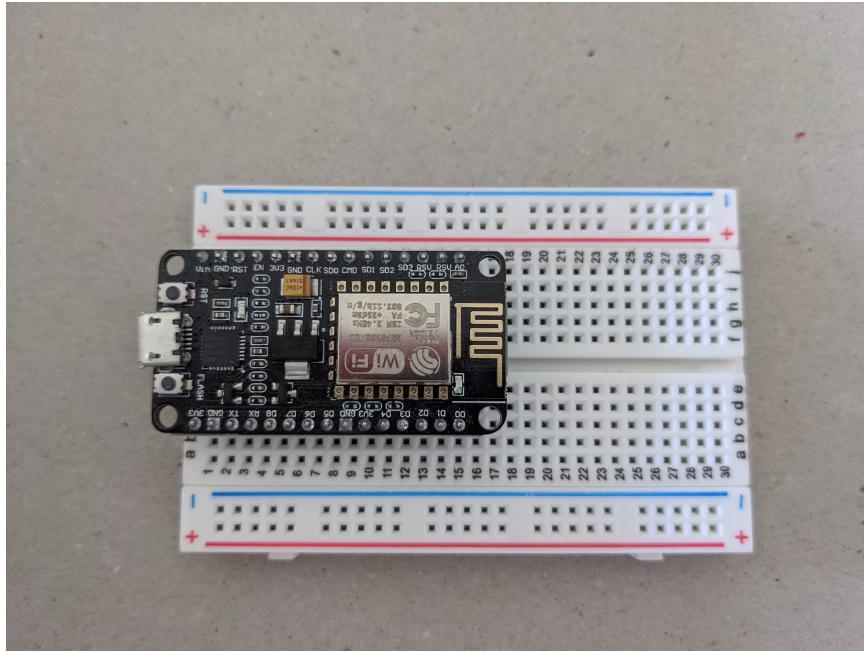


Figure 2.2: So far, so good!

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:

Connect microcontroller to LED

Connect resistor between LED and negative power rail

2.2.2 Programming the microcontroller

2.3 Review

2.4 Possible Extensions

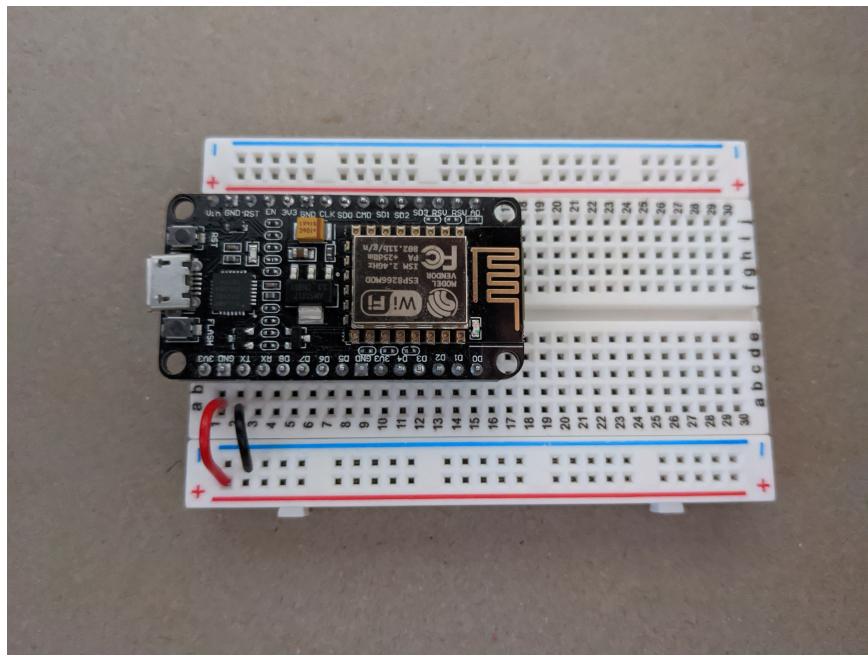


Figure 2.3: I'm absolutely POSITIVE I connected everything correctly!

Chapter 3

Project 2: Button

Chapter 4

Project 3: LED Party

Chapter 5

Project 4: Sensor

5.1 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!

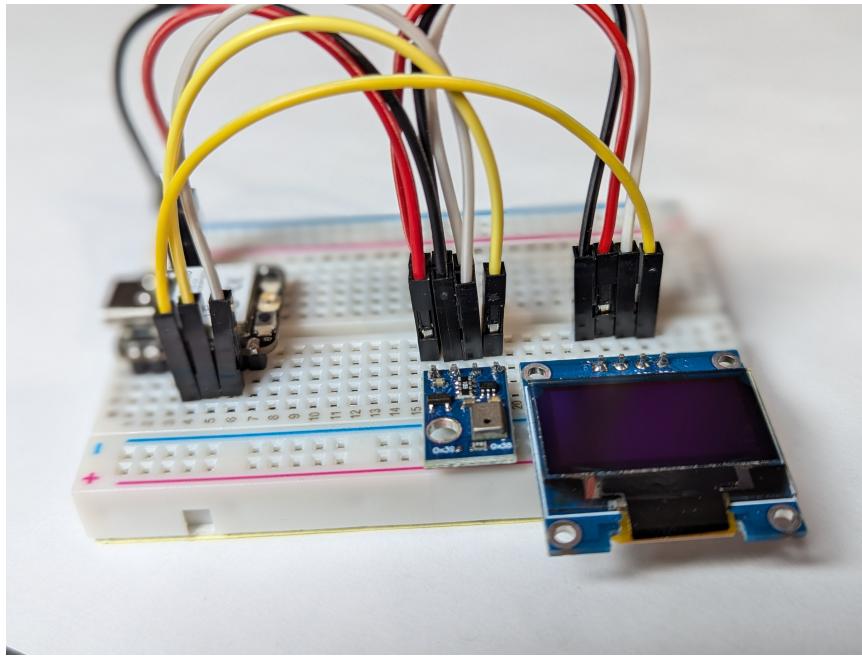


Figure 5.1: The end result should look something like this

5.2 Directions

5.2.1 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:

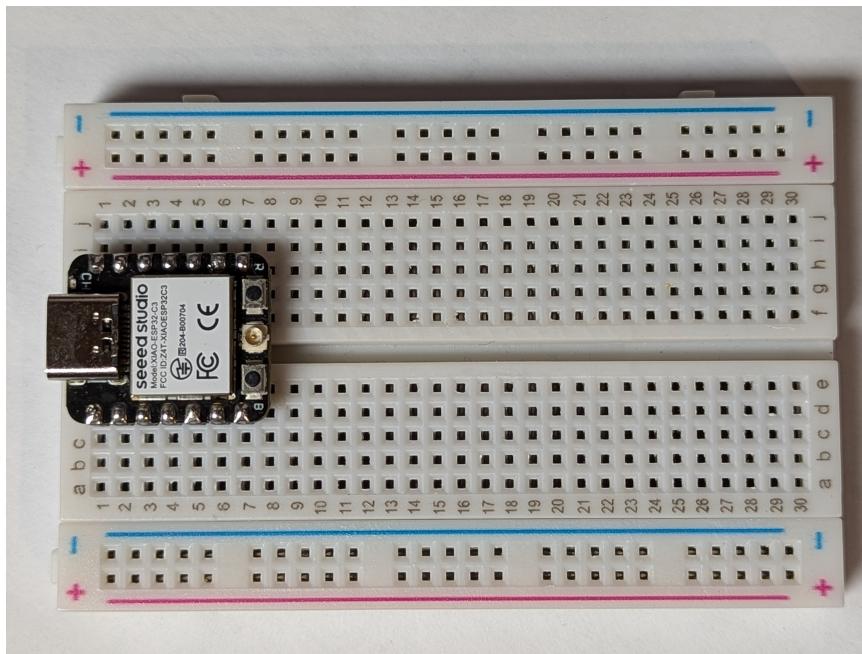


Figure 5.2: So far, so good!

Connect the necessary jumper wires

- Using a jumper wire, place one end of the 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.
- Using another jumper wire, place one end of the 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.
- Using a 3rd jumper wire, place one end of the 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.
- Finally, using a 4th jumpre wire, place one end of the 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:

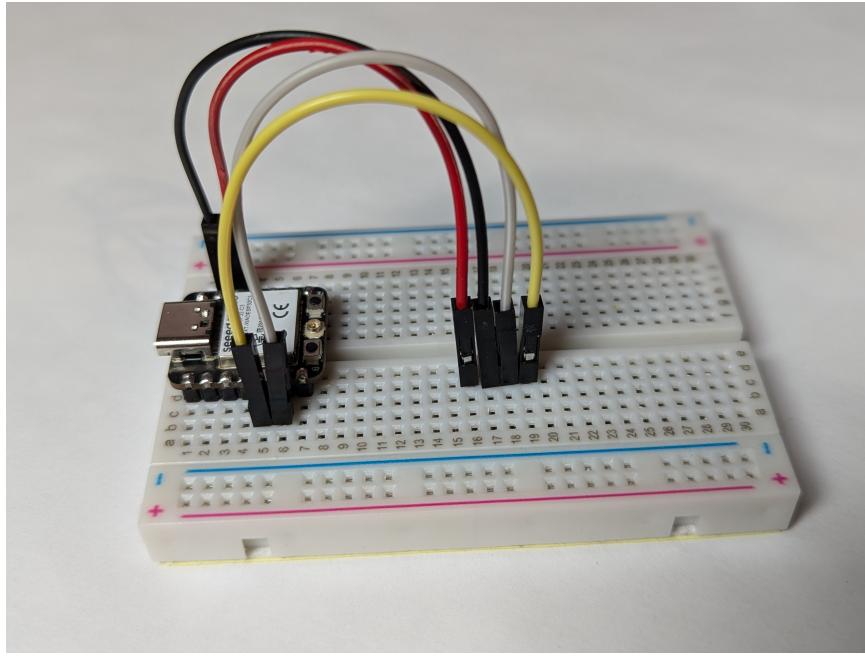


Figure 5.3: There are 4 wires that will be connected between the microcontroller and the sensor board.

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:

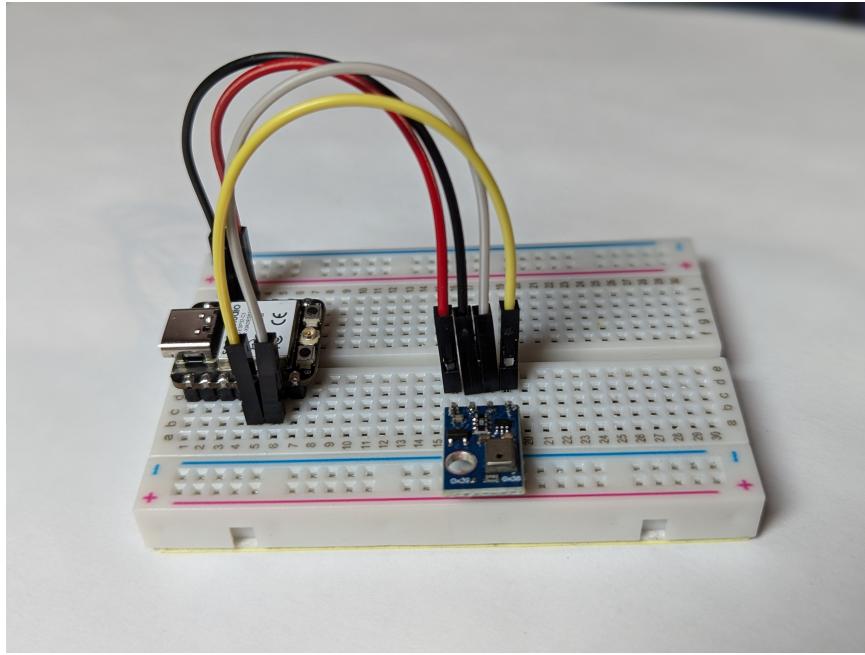


Figure 5.4: Click the button highlighted in red.

5.2.2 Programming the microcontroller

Once all of the wiring is correct, connect the USB cable to the microcontroller and go to <https://viper-ide.org/> in your computer's web browser. Click on the USB icon in the top right and choose your microcontroller from the list:

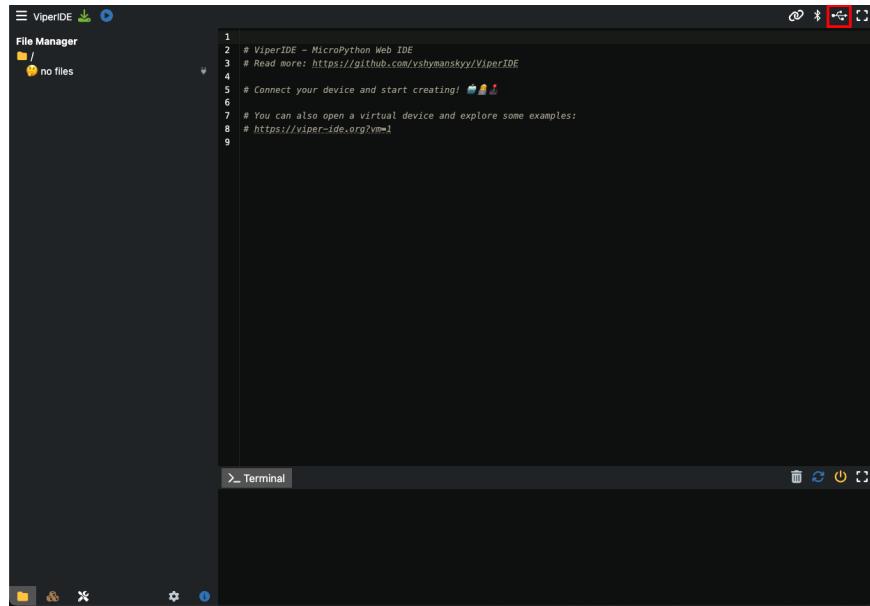


Figure 5.5: Click the button highlighted in red.

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

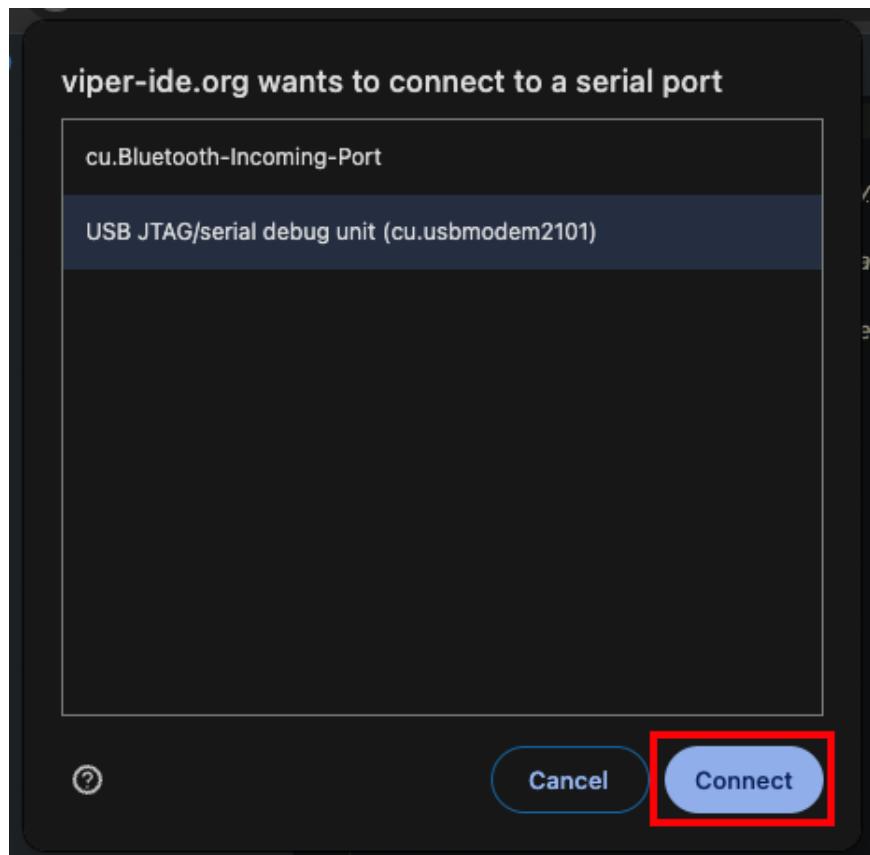


Figure 5.6: Click the button highlighted in red.

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:

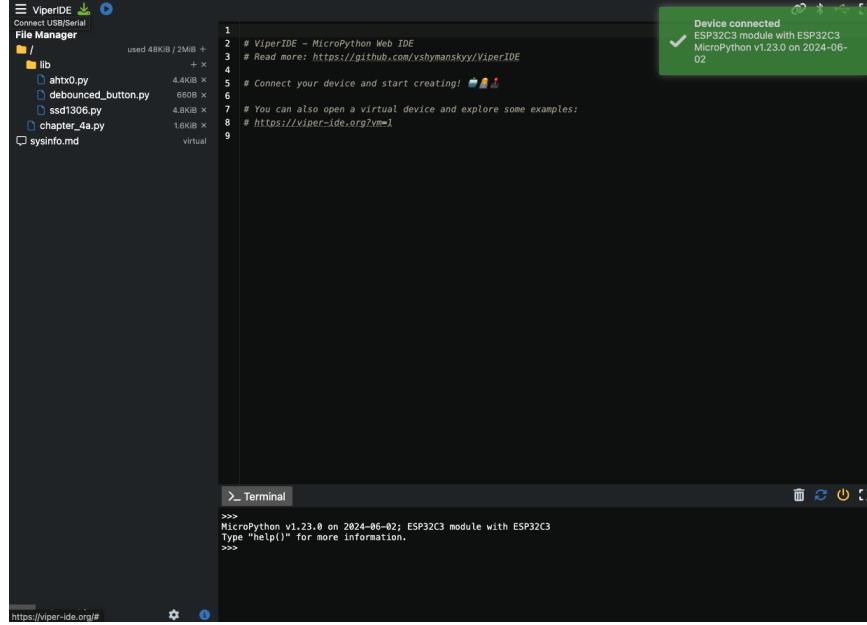


Figure 5.7: Note: update this image with the final file list

Click on the file named "chapter_4a.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:

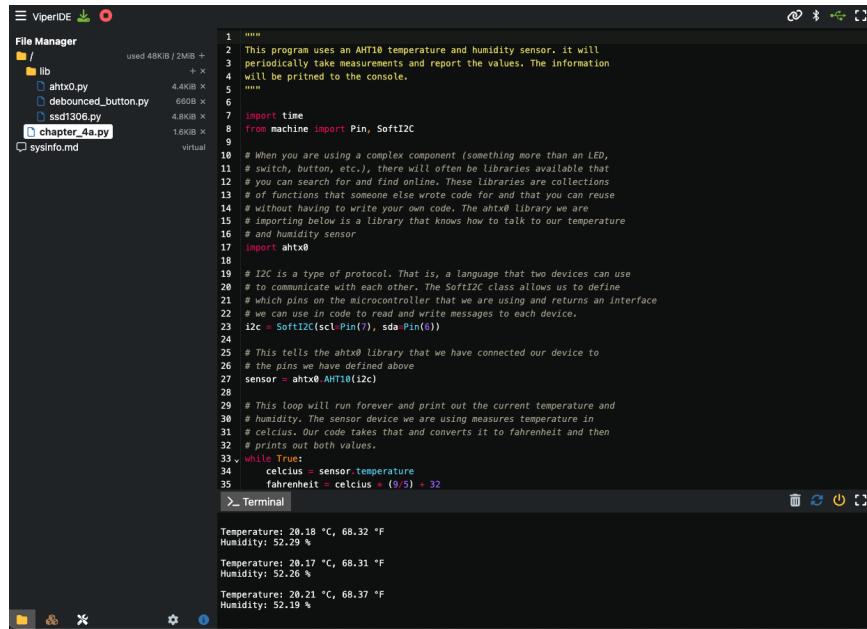


Figure 5.8: Once started, this script will run forever and output values into the terminal window. You can stop by pressing the red stop button.

5.2.3 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 shouldn't go!

- Using a jumper wire, place one end of the 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.
- Using another jumper wire, place one end of the 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.
- Using a 3rd jumper wire, place one end of the 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.
- Finally, using a 4th jumpre wire, place one end of the 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:

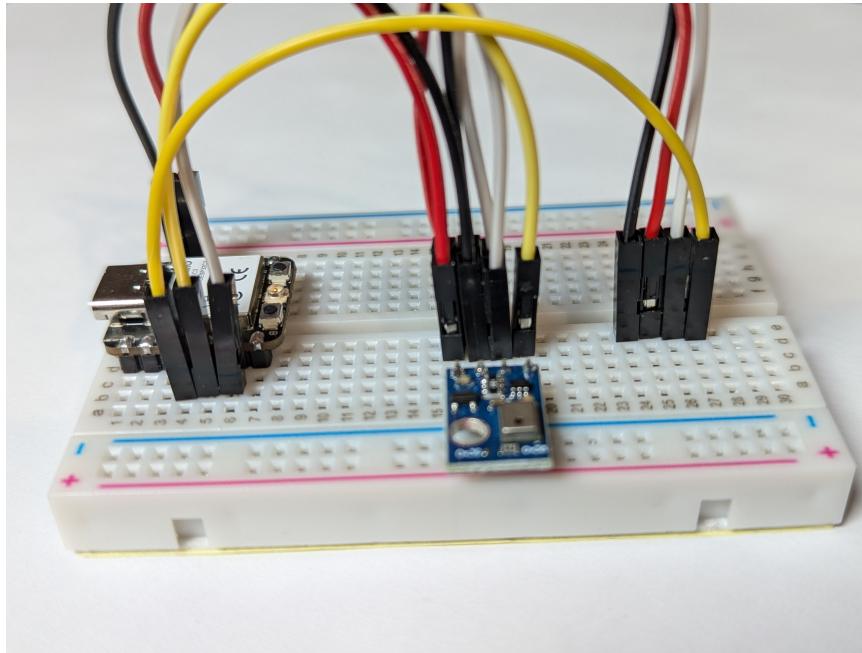


Figure 5.9: The first 3 wires are connected right behind the wires for the sensor. The last wire is connected to the microcontroller.

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:

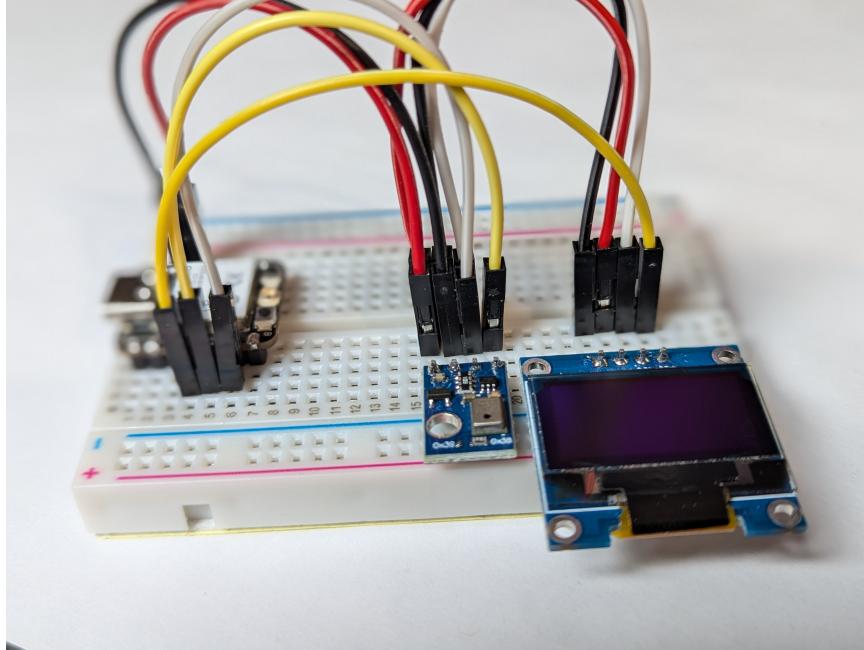


Figure 5.10: Click the button highlighted in red.

Click on the file named "chapter_4b.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:

```

ViperIDE

File Manager
  /          used 52KB / 2MB +
    lib
      ahtx0.py
      debounced_button.py
      ssd1306.py
      chapter_4a.py
      chapter_4b.py
  sysinfo.md

1 """
2 This program uses an AHT10 temperature and humidity sensor. It will
3 periodically take measurements and report the values. The information
4 will be printed to the console. This version of the code also displays
5 the information on the SSD1306 OLED screen.
6 """
7
8 import time
9 from machine import Pin, SoftI2C
10
11 # When you are using a complex component (something more than an LED,
12 # switch, button, etc.), there will often be libraries available that
13 # you can search for and find online. These libraries are collections
14 # of functions that someone else wrote code for and that you can reuse
15 # without having to write your own code. The ahtx0 library we are
16 # importing below is a library that knows how to talk to our temperature
17 # and humidity sensor
18 import ahtx0
19
20 # Here we import another library that knows how to draw things on the
21 # small screen that's included in the kit
22 import ssd1306
23
24 # I2C is a type of protocol. That is, a language that two devices can use
25 # to communicate with each other. The SoftI2C class allows us to define
26 # which pins on the microcontroller that we are using and returns an interface
27 # we can use in code to read and write messages to each device.
28 i2c_sensor = SoftI2C(scl=Pin(7), sda=Pin(6))
29
30 # create another I2C device to control our screen
31 i2c_oled = SoftI2C(scl=Pin(7), sda=Pin(5))
32
33 # This tells the ahtx0 library that we have connected our device to
34 # the pins we have defined above
35 sensor = ahtx0.AHT10(i2c_sensor)
> Terminal
Temp: 21.03 °C, 69.86 °F
Humidity: 54.61 %
Temp: 21.01 °C, 69.82 °F
Humidity: 54.33 %
Temp: 20.98 °C, 69.76 °F
Humidity: 54.18 %

```

Figure 5.11: 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.

5.3 Review

5.4 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 6

Project 5: Game

6.1 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!

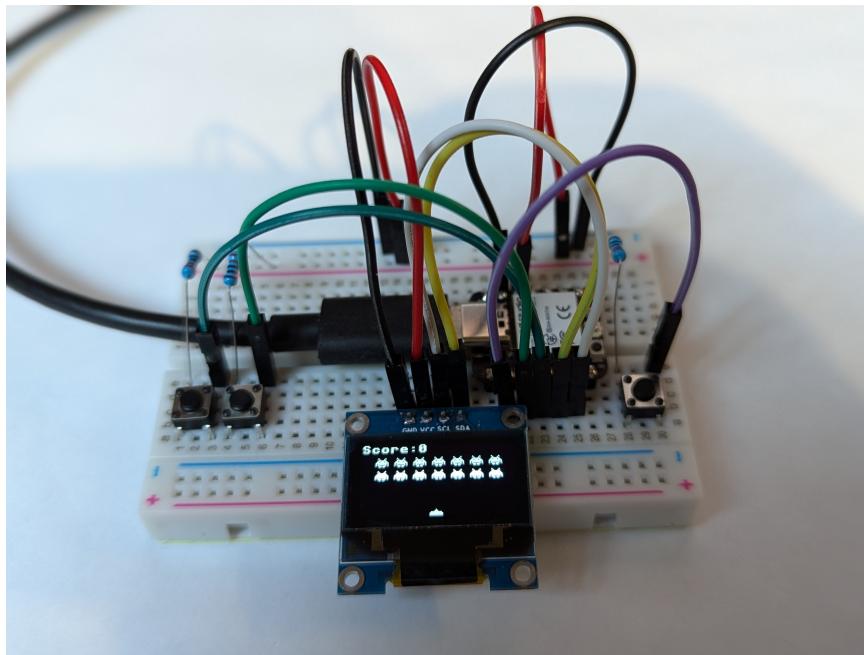


Figure 6.1: The end result should look something like this

6.2 Directions

6.2.1 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:

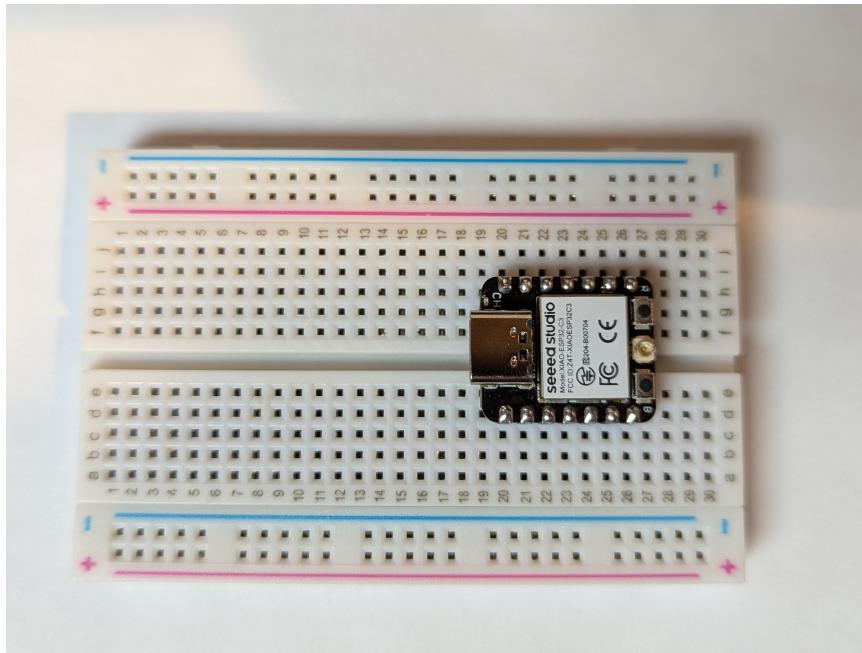


Figure 6.2: So far, so good!

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 its 4 pins into holes **A15**, **A16**, **A17**, and **A18**.

You should be left with something that looks like this:

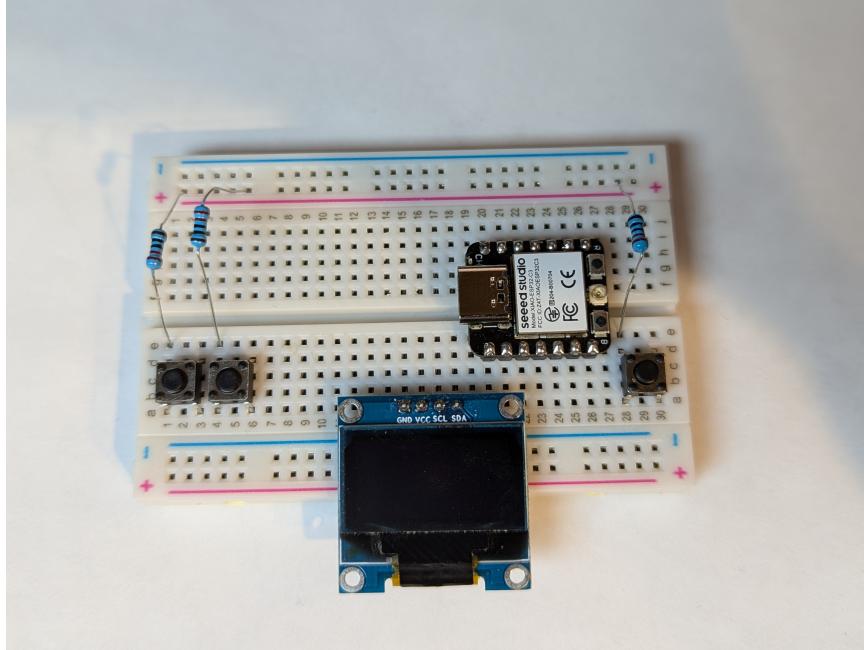


Figure 6.3: All of the components except for the jumper wires are now placed.

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:

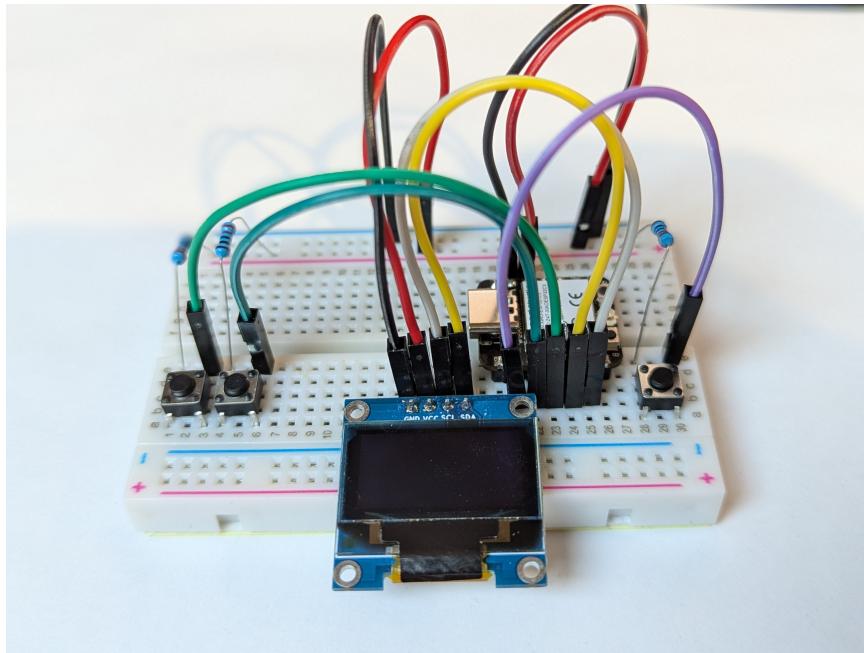


Figure 6.4: All components and wires are now installed

6.2.2 Programming the microcontroller

Once all of the wiring is correct, connect the USB cable to the microcontroller (you will want to push the cable underneath the two resistors on the left) and go to <https://viper-ide.org/> in your computer's web browser. Click on the USB icon in the top right and choose your microcontroller from the list:

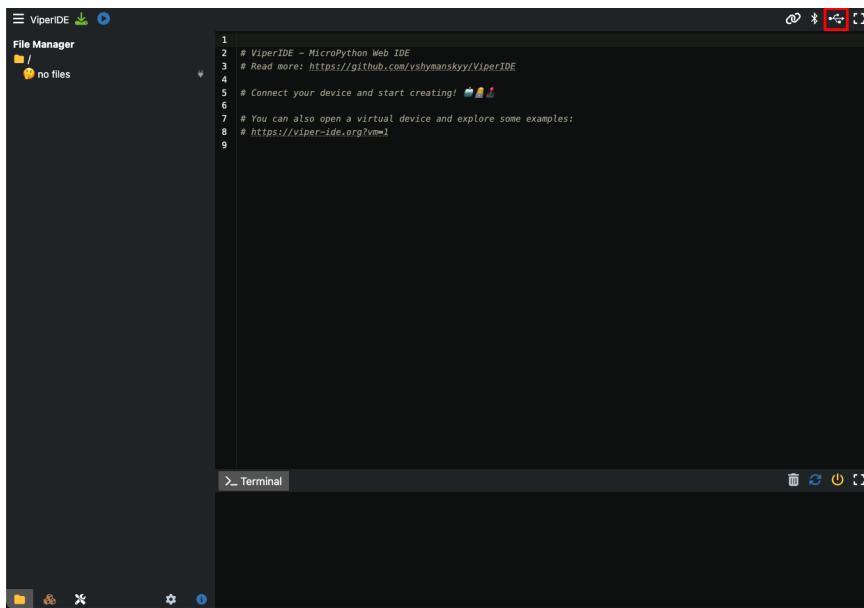


Figure 6.5: Click the button highlighted in red.

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

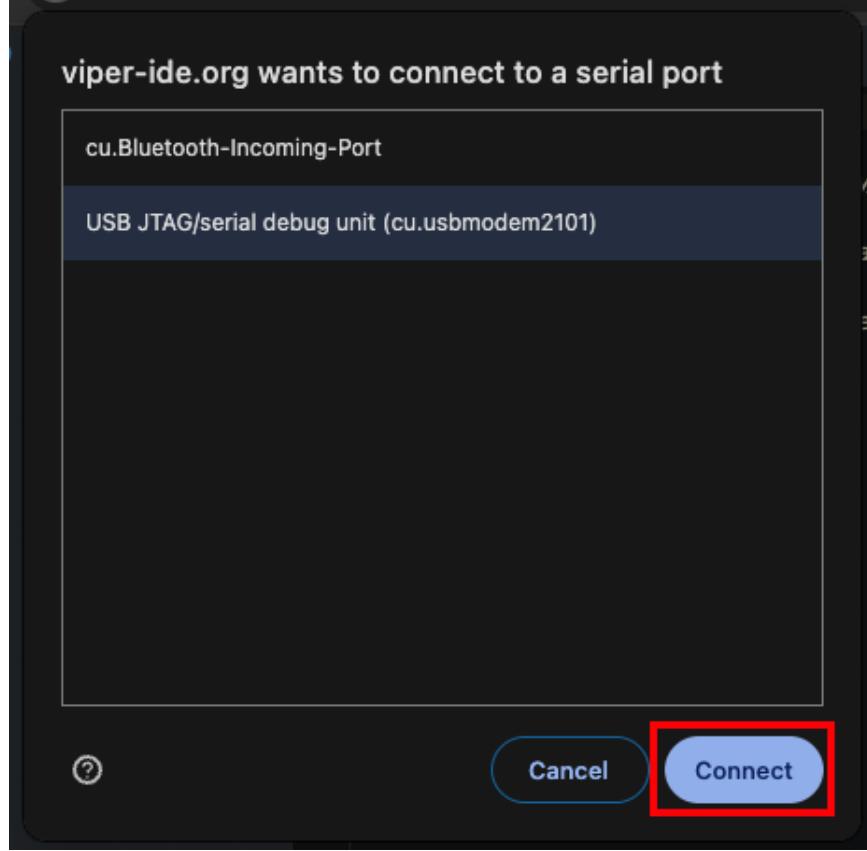


Figure 6.6: Click the button highlighted in red.

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:

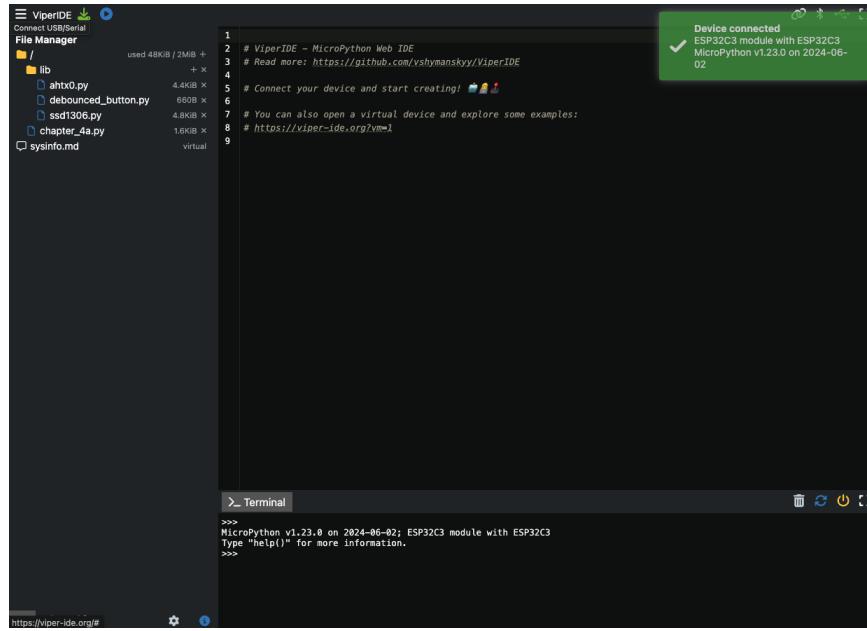


Figure 6.7: Note: update this image with the final file list

Click on the file named "chapter_5.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.

6.2.3 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 6.1: 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 6.2: 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.＼
7              ENEMY_SPRITE2]
8          self.current_sprite = 0
9          self.sprite_changed = time.ticks_ms()
10
11     @property
12     def hitbox(self):
13         return (self.x, self.y, 16, 8)
14
15     def move(self):
16         pass
17
18     def draw(self):
19         now = time.ticks_ms()
20         if now - self.sprite_changed > 1000:
21             self.current_sprite += 1
22             self.current_sprite %= len(self.sprites)
23             self.sprite_changed = now
24         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.py` 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 than having to draw out each pixel in a loop.

The Ship class

Listing 6.3: 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 6.4: 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 6.5: The main function

```

1  def main():
2      try:
3          ship = Ship(64, 64)
4          for y in range(12, 36, 12):
5              for x in range(10, 118, 16):
6                  enemies.append(Enemy(x, y))
7          while game_on:
8              game_loop(ship)
9      except KeyboardInterrupt:
10         pass
11
12 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.

6.3 Review

6.4 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 7

Project 6: Sound

Chapter 8

Project 7: Chat

Appendix A

Electronics Essentials

Appendix B

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 dissect and explain what each section does below:

Listing B.1: An example Python script

```
1 def show(message, repeat=1):
2     """This function prints the given message to the
3     console as many 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
```

Another feature that you'll see used often in Python are classes. Classes are a convenient 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 B.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 also 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.