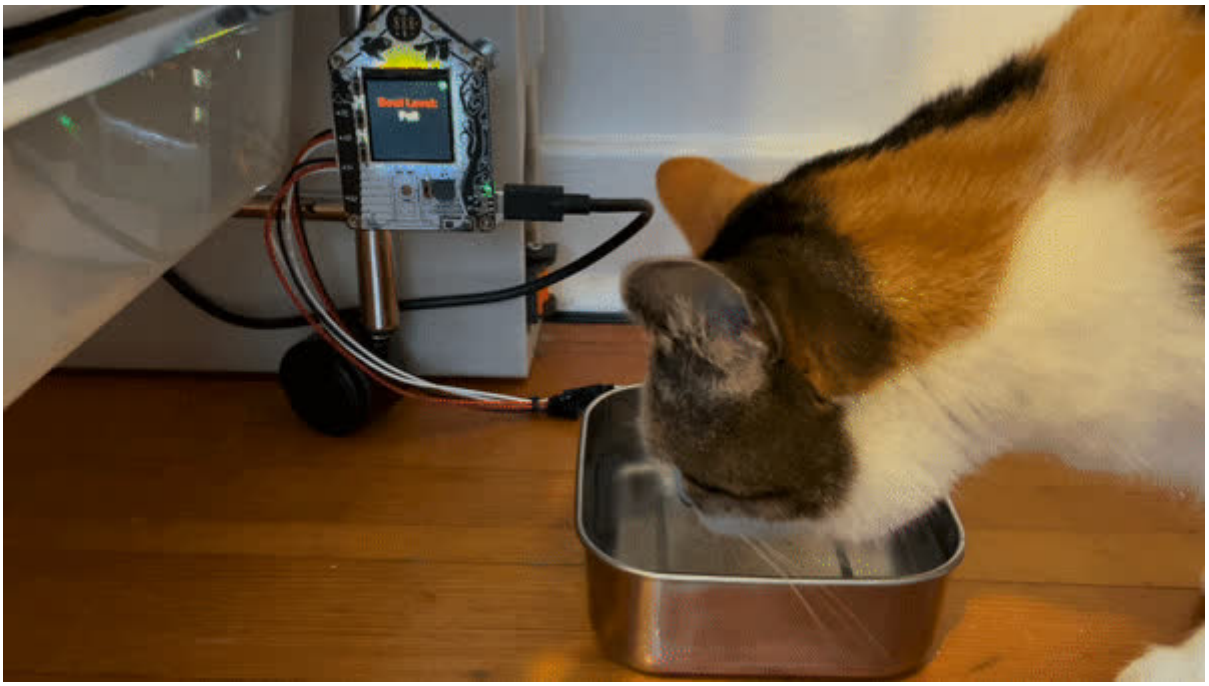
Pet Bowl Water Level Sensing

Created by Melissa LeBlanc-Williams

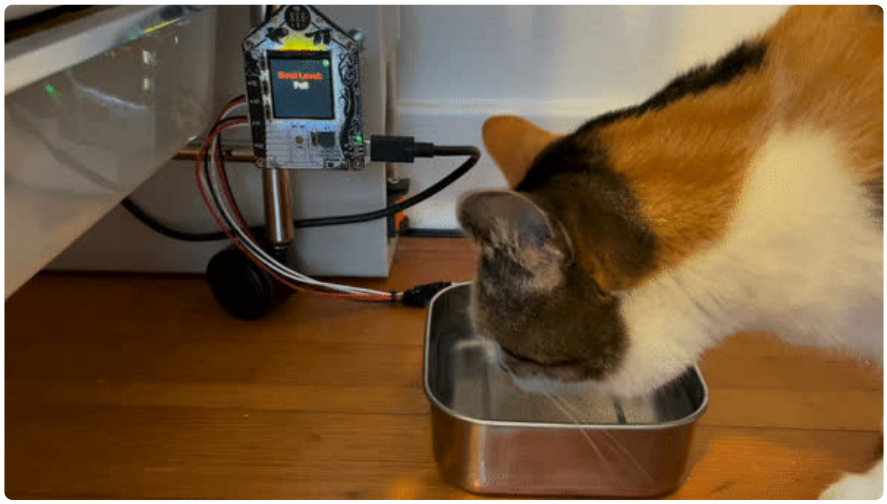


https://learn.adafruit.com/pet-bowl-water-level-sensing-with-the-funhouse-and-home-assistant

Last updated on 2024-11-29 10:06:18 AM EST

Table of Contents

Overview

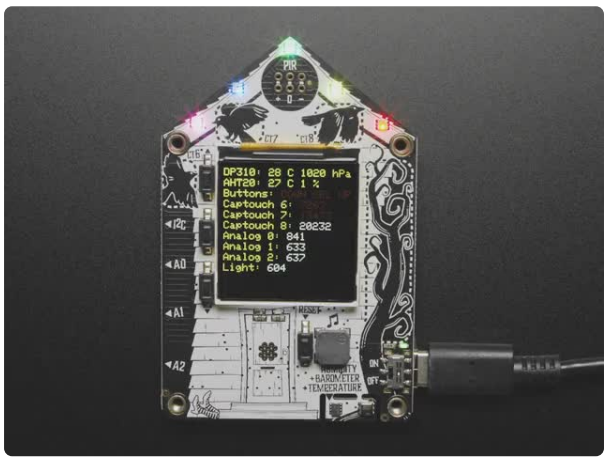


With the FunHouse and Home Assistant, you can automatically be alerted if your pet's bowl gets too low. For this project we are using the Simple Water Sensor. Even though it is designed to be a digital sensor, it will actually work in analog mode to give you a little more granularity. In this way, you can tell if your pet's water level is good, low, or needs water added.

In order to mitigate the effects of running electricity through water, the sensor is only on for a very short period of time to take a reading and turns back off. This is accomplished by using 2 of the ports on the FunHouse. One supplies the voltage to the sensor through the GPIO line and the other takes the reading.

We'll go over how to get your sensor set up and configure Home Assistant to read the status of the sensor and also send you notifications within the Home Assistant environment.

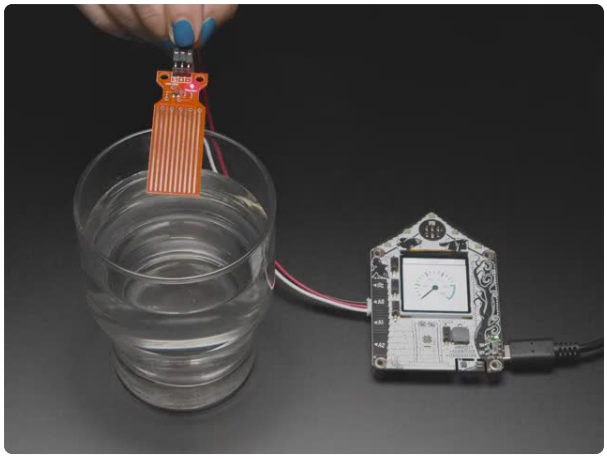
Parts



[Adafruit FunHouse - WiFi Home Automation Development Board]

Home is where the heart is...it's also where we keep all our electronic bits. So why not wire it up with sensors and actuators to turn our house into an electronic wonderland....

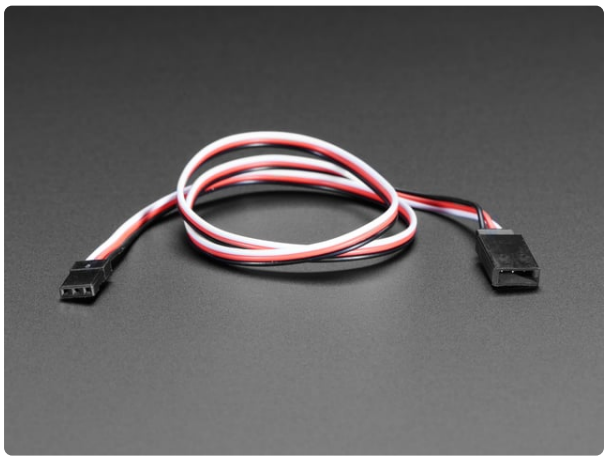
https://www.adafruit.com/product/4985



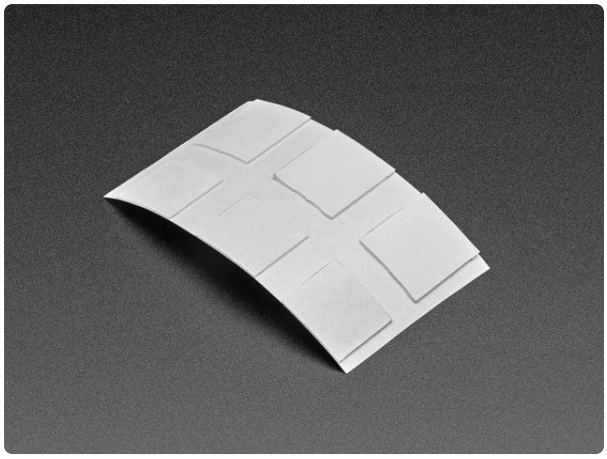
[Simple Water Detection Sensor with Digital Output]
Keep wet things wet and dry things dry by detecting when the dry things get wet by accident! This palm-sized cherry-red water sensor is simple and easy to implement in your...
https://www.adafruit.com/product/4965

You'll need 2 of these STEMMA adapters.

[STEMMA JST PH 2mm 3-Pin to Female Socket Cable - 200mm]
This cable will let you turn a JST PH 3-pin cable port into 3 individual wires with high-quality 0.1" female header sockets on the end. We're carrying these to match up with...
https://www.adafruit.com/product/3894



[Servo Extension Cable - 50cm / 19.5" long]
Stretch out your servo connections with this flexible servo extension cord. It has a 3 pin shrouded "male" connection to plug your servo into and then, 50cm later, a 3 pin...
https://www.adafruit.com/product/973



Clear Adhesive Squares - 6 pack

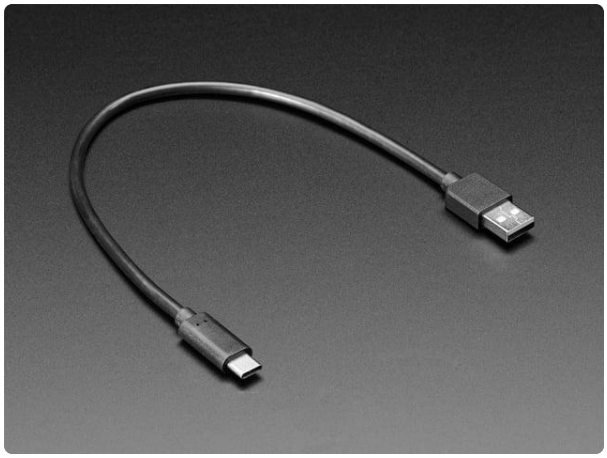
UGlu Dashes are perfect for a variety of small projects. These adhesive squares provide a stronger bond to most surfaces and are cleaner and easier to...

https://www.adafruit.com/product/4813

Mini Magnet Feet for RGB LED Matrices (Pack of 4)

Got a glorious RGB Matrix project you want to mount and display in your workspace or home? If you have one of the matrix panels listed below, you'll need a pack of these...

https://www.adafruit.com/product/4631



USB Type A to Type C Cable - 1ft - 0.3 meter

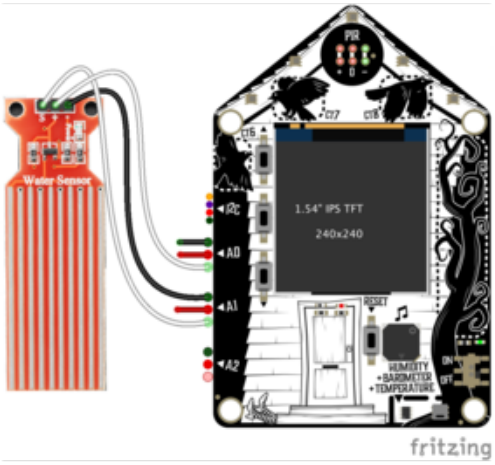
As technology changes and adapts, so does Adafruit. This USB Type A to Type C cable will help you with the transition to USB C, even if you're still...

https://www.adafruit.com/product/4473

Wiring

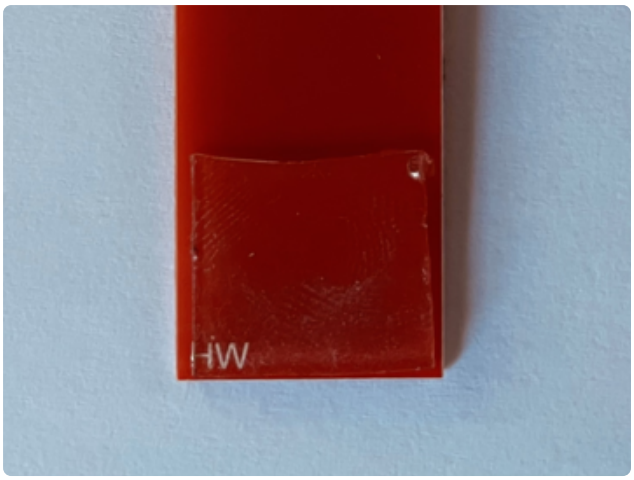
Wiring the FunHouse to the water sensor is pretty straight forward. The only tricky bit we are doing here is we are using 2 JST ports to read the sensor. One of the JST ports is used for reading the sensor and the other port is powering the sensor with its signal wire. This is so the sensor isn't on all the time, which can lead to electrolyzing the traces fairly quickly.

Since this is a water sensor and we're dealing with electronics, we recommend connecting a servo extension cable between the JST cables and the sensor.

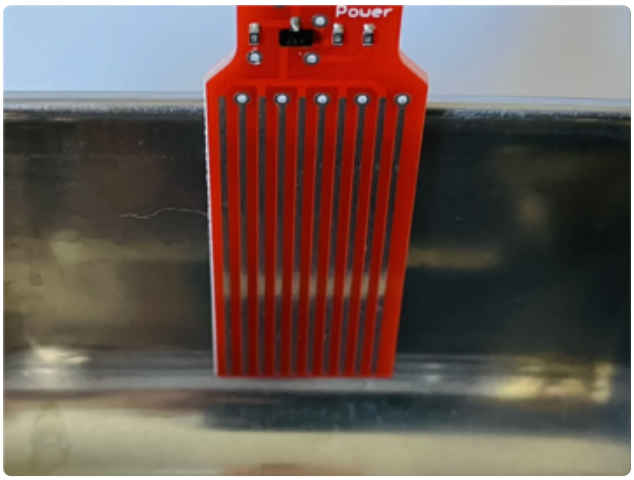


Connect the **white wire** from the JST connector of **A0** to the positive or + pin on the water sensor.
Connect the **black wire** from the JST connector of **A1** to the ground or - pin on the water sensor.
Connect the **white wire** from the JST connector of **A1** to the signal or S pin on the water sensor.
Leave any remaining wires unconnected.

Attaching the Sensor

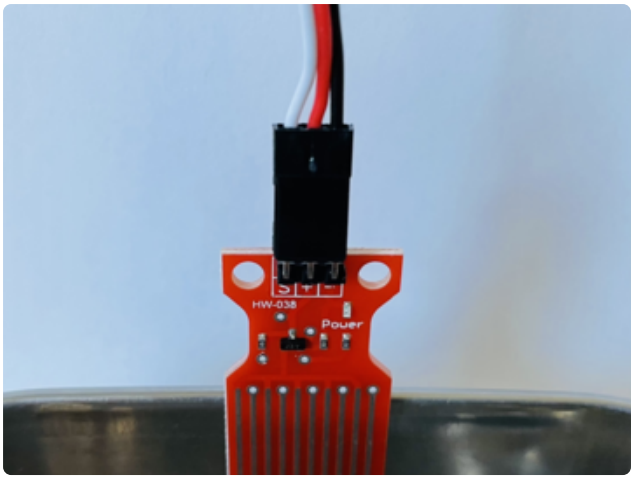


Place some double-sided adhesive to the back side of the sensor.



Attach the sensor to the side of the bowl. If it has trouble sticking due to the curve in the bowl, you can add more adhesive.

You'll want to align the bottom edge around where you would want to be alerted, so if your pet likes it changed more frequently and doesn't drink much, you may want to mount it a bit higher.



Plug the servo extension cable into the sensor with the black wire towards the `-` and the white wire towards the `S` as shown in the image.

CircuitPython

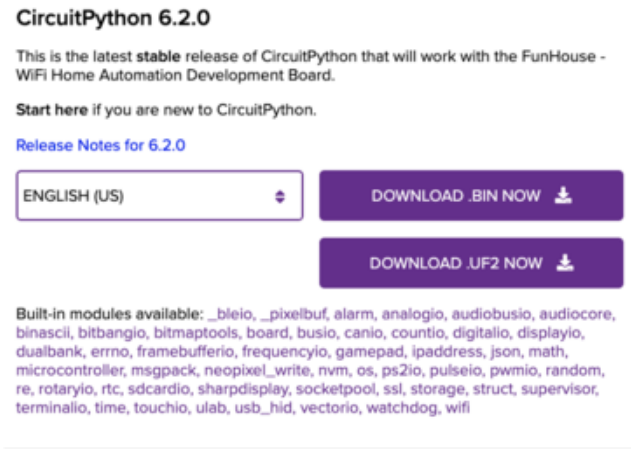
CircuitPython (https://adafru.it/tB7) is a derivative of MicroPython (https://adafru.it/f9W) designed to simplify experimentation and education on low-cost microcontrollers. It makes it easier than ever to get prototyping by requiring no upfront desktop software downloads. Simply copy and edit files on the **CIRCUITPY** drive to iterate.

Set Up CircuitPython

Follow the steps to get CircuitPython installed on your FunHouse.

Download the latest CircuitPython for your board from circuitpython.org

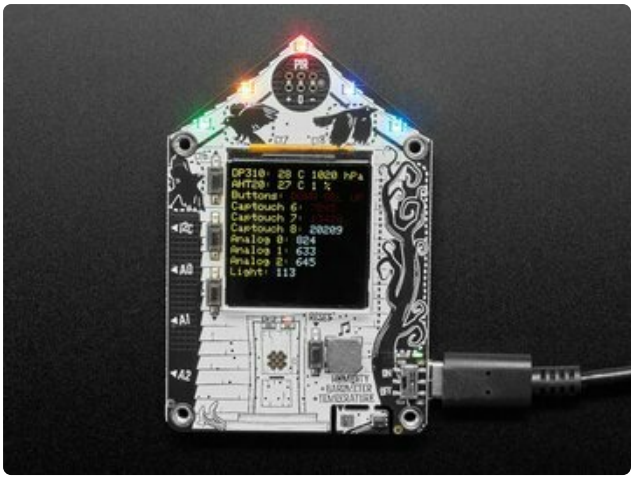
https://adafru.it/RKB



Click the link above and download the latest .BIN and .UF2 file

(depending on how you program the ESP32S2 board you may need one or the other, might as well get both)

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



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

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

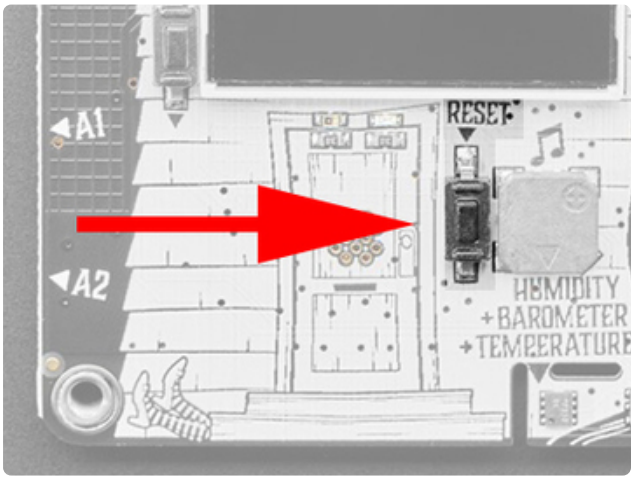
Option 1 - Load with UF2 Bootloader

This is by far the easiest way to load CircuitPython. **However it requires your board has the UF2 bootloader installed. Some early boards do not (we hadn't written UF2 yet!) - in which case you can load using the built in ROM bootloader.**

Still, try this first!

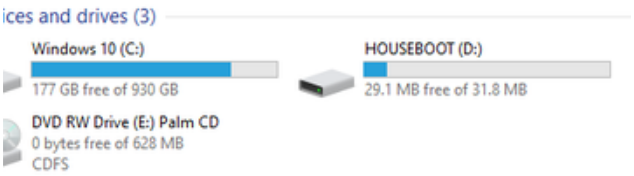
Try Launching UF2 Bootloader

Loading CircuitPython by drag-n-drop UF2 bootloader is the easier way and we recommend it.



Launch UF2 by **double-clicking** the Reset button (the one next to the USB C port). You may have to try a few times to get the timing right.

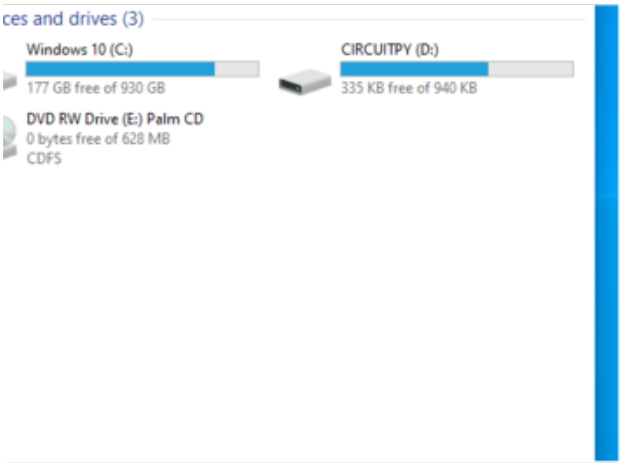
About a half second pause between clicks while the DotStars are purple seems to work well.



If the UF2 bootloader is installed, you will see a new disk drive appear called **HOUSEBOOT**

Copy the **UF2** file you downloaded at the first step of this tutorial onto the **HOUSEBOOT** drive

If you're using Windows and you get an error at the end of the file copy that says **Error from the file copy, Error 0x800701B1: A device which does not exist was specified.** You can ignore this error, the bootloader sometimes disconnects without telling Windows, the install completed just fine and you can continue. If its really annoying, you can also upgrade the bootloader (the latest version of the UF2 bootloader fixes this warning) (https://adafru.it/RLc)



Your board should auto-reset into CircuitPython, or you may need to press reset. A **CIRCUITPY** drive will appear. You're done! Go to the next pages.

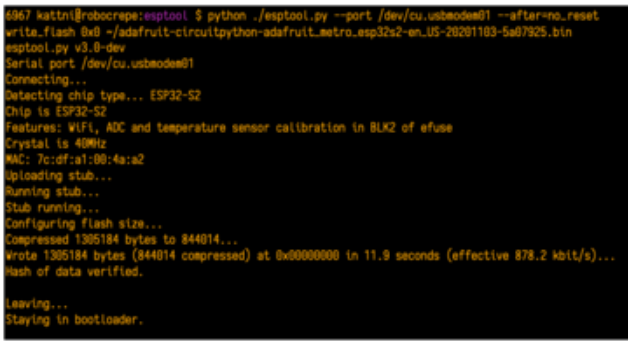
Option 2 - Use Chrome Browser To Upload BIN file

> You will need to do a full erase prior to uploading new firmware.

The next best option is to try using the Chrome-browser version of esptool we have written. This is handy if you don't have Python on your computer, or something is really weird with your setup that makes esptool not run (which happens sometimes and isn't worth debugging!) You can follow along on the Install UF2 Bootloader (https://adafru.it/RLc) page and either load the UF2 bootloader and then come back to Option 1 on this page, or you can download the CircuitPython BIN file directly using the tool in the same manner as the bootloader.

Option 3 - Use esptool to load BIN file

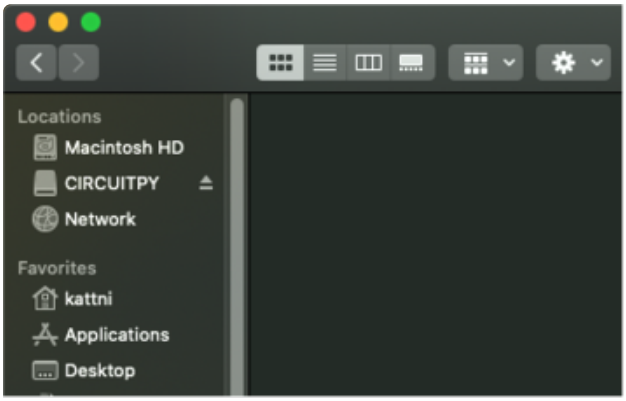
For more advanced users, you can upload with **esptool** to the ROM (hardware) bootloader instead!



Follow the initial steps found in the Run esptool and check connection section of the Install UF2 Bootloader page (https://adafru.it/RLc) to verify your environment is set up, your board is successfully connected, and which port it's using.

In the final command to write a binary file to the board, replace the port with your port, and replace "firmware.bin" with the the file you downloaded above.

The output should look something like the output in the image.



Press reset to exit the bootloader.

Your **CIRCUITPY** drive should appear!

You're all set! Go to the next pages.

CircuitPython Internet Test

One of the great things about the ESP32 is the built-in WiFi capabilities. This page covers the basics of getting connected using CircuitPython.

The first thing you need to do is update your **code.py** to the following. Click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, and copy the **entire lib folder** and the **code.py** file to your **CIRCUITPY** drive.

```
# SPDX-FileCopyrightText: 2020 Brent Rubell for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import os
import ipaddress
import ssl
import wifi
import socketpool
import adafruit_requests

# URLs to fetch from
TEXT_URL = "http://wifitest.adafruit.com/testwifi/index.html"
JSON_QUOTES_URL = "https://www.adafruit.com/api/quotes.php"
JSON_STARS_URL = "https://api.github.com/repos/adafruit/circuitpython"

print("ESP32-S2 WebClient Test")

print(f"My MAC address: {[hex(i) for i in wifi.radio.mac_address]}")

print("Available WiFi networks:")
for network in wifi.radio.start_scanning_networks():
    print("\t%s\t\tRSSI: %d\tChannel: %d" % (str(network.ssid, "utf-8"),
                                              network.rssi, network.channel))
wifi.radio.stop_scanning_networks()

print(f"Connecting to {os.getenv('CIRCUITPY_WIFI_SSID')}")
wifi.radio.connect(os.getenv("CIRCUITPY_WIFI_SSID"),
os.getenv("CIRCUITPY_WIFI_PASSWORD"))
print(f"Connected to {os.getenv('CIRCUITPY_WIFI_SSID')}")
print(f"My IP address: {wifi.radio.ipv4_address}")

ping_ip = ipaddress.IPv4Address("8.8.8.8")
```

```
ping = wifi.radio.ping(ip=ping_ip)

# retry once if timed out
if ping is None:
    ping = wifi.radio.ping(ip=ping_ip)

if ping is None:
    print("Couldn't ping 'google.com' successfully")
else:
    # convert s to ms
    print(f"Pinging 'google.com' took: {ping * 1000} ms")

pool = socketpool.SocketPool(wifi.radio)
requests = adafruit_requests.Session(pool, ssl.create_default_context())

print(f"Fetching text from {TEXT_URL}")
response = requests.get(TEXT_URL)
print("-" * 40)
print(response.text)
print("-" * 40)

print(f"Fetching json from {JSON_QUOTES_URL}")
response = requests.get(JSON_QUOTES_URL)
print("-" * 40)
print(response.json())
print("-" * 40)

print()

print(f"Fetching and parsing json from {JSON_STARS_URL}")
response = requests.get(JSON_STARS_URL)
print("-" * 40)
print(f"CircuitPython GitHub Stars: {response.json()['stargazers_count']}")
print("-" * 40)

print("Done")
```

Your **CIRCUITPY** drive should resemble the following.

To get connected, the next thing you need to do is update the **settings.toml** file.

The settings.toml File

We expect people to share tons of projects as they build CircuitPython WiFi widgets. What we want to avoid is people accidentally sharing their passwords or secret tokens and API keys. So, we designed all our examples to use a **settings.toml** file, that is on your **CIRCUITPY** drive, to hold secret/private/custom data. That way you can share your main project without worrying about accidentally sharing private stuff.

If you have a fresh install of CircuitPython on your board, the initial **settings.toml** file on your **CIRCUITPY** drive is empty.

To get started, you can update the **settings.toml** on your **CIRCUITPY** drive to contain the following code.

```
# SPDX-FileCopyrightText: 2023 Adafruit Industries
#
# SPDX-License-Identifier: MIT

# This is where you store the credentials necessary for your code.
# The associated demo only requires WiFi, but you can include any
# credentials here, such as Adafruit IO username and key, etc.
CIRCUITPY_WIFI_SSID = "your-wifi-ssid"
CIRCUITPY_WIFI_PASSWORD = "your-wifi-password"
```

This file should contain a series of Python variables, each assigned to a string. Each variable should describe what it represents (say `wifi_ssid`), followed by an = (equals sign), followed by the data in the form of a Python string (such as `"my-wifi-password"` including the quote marks).

At a minimum you'll need to add/update your WiFi SSID and WiFi password, so do that now!

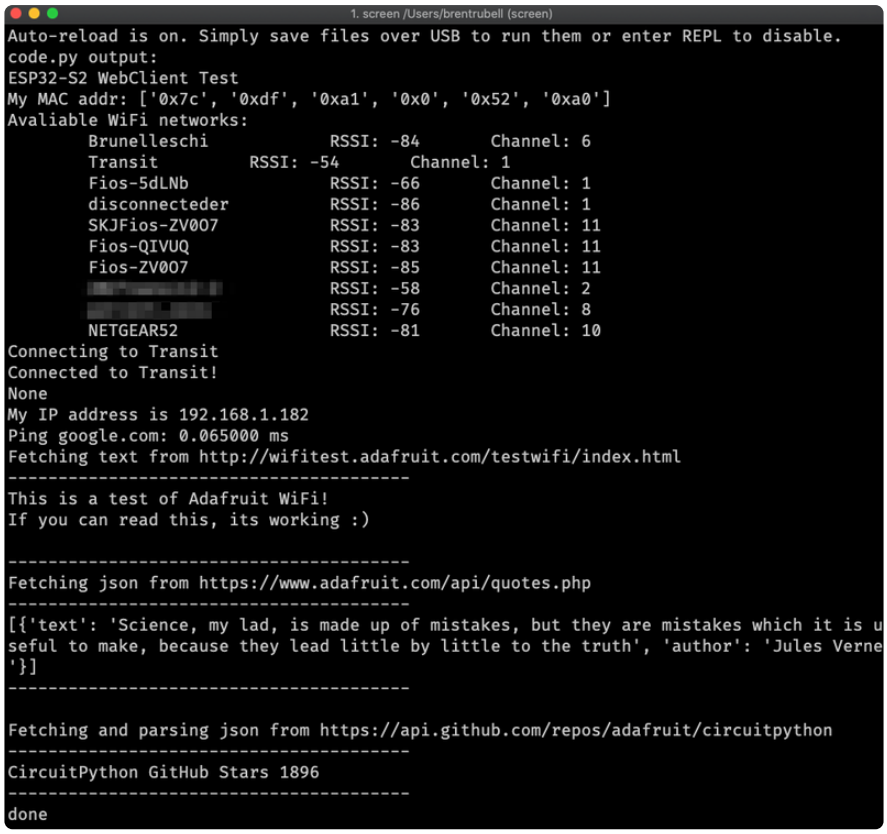
As you make projects you may need more tokens and keys, just add them one line at a time. See for example other tokens such as one for accessing GitHub or the Hackaday API. Other non-secret data like your timezone can also go here.

For the correct time zone string, look at http://worldtimeapi.org/timezones (https://adafru.it/EcP) and remember that if your city is not listed, look for a city in the same time zone, for example Boston, New York, Philadelphia, Washington DC, and Miami are all on the same time as New York.

Of course, don't share your **settings.toml** - keep that out of GitHub, Discord or other project-sharing sites.

> Don't share your settings.toml file! It has your passwords and API keys in it!

If you connect to the serial console, you should see something like the following:



In order, the example code...

Checks the ESP32's MAC address.

```
print(f"My MAC address: {[hex(i) for i in wifi.radio.mac_address]}")
```

Performs a scan of all access points and prints out the access point's name (SSID), signal strength (RSSI), and channel.

```
print("Available WiFi networks:")
for network in wifi.radio.start_scanning_networks():
    print("\t%s\t\tRSSI: %d\tChannel: %d" % (str(network.ssid, "utf-8"),
                                              network.rssi, network.channel))
wifi.radio.stop_scanning_networks()
```

Connects to the access point you defined in the **settings.toml** file, and prints out its local IP address.

```
print(f"Connecting to {os.getenv('WIFI_SSID')}")
wifi.radio.connect(os.getenv("WIFI_SSID"), os.getenv("WIFI_PASSWORD"))
print(f"Connected to {os.getenv('WIFI_SSID')}")
print(f"My IP address: {wifi.radio.ipv4_address}")
```

Attempts to ping a Google DNS server to test connectivity. If a ping fails, it returns `None`. Initial pings can sometimes fail for various reasons. So, if the initial ping is successful (`is not None`), it will print the echo speed in ms. If the initial ping fails, it

will try one more time to ping, and then print the returned value. If the second ping fails, it will result in `"Ping google.com: None ms"` being printed to the serial console. Failure to ping does not always indicate a lack of connectivity, so the code will continue to run.

```
ping_ip = ipaddress.IPv4Address("8.8.8.8")
ping = wifi.radio.ping(ip=ping_ip) * 1000
if ping is not None:
    print(f"Ping google.com: {ping} ms")
else:
    ping = wifi.radio.ping(ip=ping_ip)
    print(f"Ping google.com: {ping} ms")
```

The code creates a socketpool using the wifi radio's available sockets. This is performed so we don't need to re-use sockets. Then, it initializes a a new instance of the requests (https://adafru.it/E9o) interface - which makes getting data from the internet really really easy.

```
pool = socketpool.SocketPool(wifi.radio)
requests = adafruit_requests.Session(pool, ssl.create_default_context())
```

To read in plain-text from a web URL, call `requests.get` - you may pass in either a http, or a https url for SSL connectivity.

```
print(f"Fetching text from {TEXT_URL}")
response = requests.get(TEXT_URL)
print("-" * 40)
print(response.text)
print("-" * 40)
```

Requests can also display a JSON-formatted response from a web URL using a call to `requests.get`.

```
print(f"Fetching json from {JSON_QUOTES_URL}")
response = requests.get(JSON_QUOTES_URL)
print("-" * 40)
print(response.json())
print("-" * 40)
```

Finally, you can fetch and parse a JSON URL using `requests.get`. This code snippet obtains the `stargazers_count` field from a call to the GitHub API.

```
print(f"Fetching and parsing json from {JSON_STARS_URL}")
response = requests.get(JSON_STARS_URL)
print("-" * 40)
print(f"CircuitPython GitHub Stars: {response.json()['stargazers_count']}")
print("-" * 40)
```

OK you now have your ESP32 board set up with a proper **settings.toml** file and can connect over the Internet. If not, check that your **settings.toml** file has the right SSID and password and retrace your steps until you get the Internet connectivity working!

IPv6 Networking

Starting in CircuitPython 9.2, IPv6 networking is available on most Espressif wifi boards. Socket-using libraries like **adafruit_requests** and **adafruit_ntp** will need to be updated to use the new APIs and for now can only connect to services on IPv4.

IPv6 connectivity & privacy

IPv6 addresses are divided into many special kinds, and many of those kinds (like those starting with **FC**, **FD**, **FE**) are private or local; Addresses starting with other prefixes like **2002:** and **2001:** are globally routable. In 2024, far from all ISPs and home networks support IPv6 internet connectivity. For more info consult resources like Wikipedia (https://adafru.it/1a4z). If you're interested in global IPv6 connectivity you can use services like Hurricane Electric (https://adafru.it/1a4A) to create an "IPv6 tunnel" (free as of 2024, but requires expertise and a compatible router or host computer to set up)

It's also important to be aware that, as currently implemented by Espressif, there are privacy concerns especially when these devices operate on the global IPv6 network: The device's unique identifier (its EUI-64 or MAC address) is used by default as part of its IPv6 address. This means that the device identity can be tracked across multiple networks by any service it connects to.

Enable IPv6 networking

Due to the privacy consideration, IPv6 networking is not automatically enabled. Instead, it must be explicitly enabled by a call to `start_dhcp_client` with the `ipv6=True` argument specified:

```
wifi.start_dhcp_client(ipv6=True)
```

Check IP addresses

The read-only `addresses` property of the `wifi.radio` object holds all addresses, including IPv4 and IPv6 addresses:

```
&gt;&gt;&gt; wifi.radio.addresses
('FE80::7EDF:A1FF:FE00:518C', 'FD5F:3F5C:FE50:0:7EDF:A1FF:FE00:518C', '10.0.3.96')
```

The `wifi.radio.dns` servers can be IPv4 or IPv6:

```
&gt;&gt;&gt; wifi.radio.dns
('FD5F:3F5C:FE50::1',)
```

```
&gt;&gt;&gt; wifi.radio.dns = ("1.1.1.1",)
&gt;&gt;&gt; wifi.radio.dns
('1.1.1.1',)
```

Ping v6 networks

`wifi.radio.ping` accepts v6 addresses and names:

```
&gt;&gt;&gt; wifi.radio.ping("google.com")
0.043
&gt;&gt;&gt; wifi.radio.ping("ipv6.google.com")
0.048
```

Create & use IPv6 sockets

Use the address family `socket.AF_INET6`. After the socket is created, use methods like `connect`, `send`, `recfrom_into`, etc just like for IPv4 sockets. This code snippet shows communicating with a private-network NTP server; this IPv6 address will not work on your network:

```
&gt;&gt;&gt; ntp_addr = ("fd5f:3f5c:fe50::20e", 123)
&gt;&gt;&gt; PACKET_SIZE = 48
&gt;&gt;&gt;
&gt;&gt;&gt; buf = bytearray(PACKET_SIZE)
&gt;&gt;&gt; with socket.socket(socket.AF_INET6, socket.SOCK_DGRAM) as s:
...     s.settimeout(1)
...     buf[0] = 0b0010_0011
...     s.sendto(buf, ntp_addr)
...     print(s.recvfrom_into(buf))
...     print(buf)
...
48
(48, ('fd5f:3f5c:fe50::20e', 123))
bytearray(b'$\x01\x03\xeb\x00\x00\x00\x00\x00\x00\x00GGPS\x00\xeaA0h\x07s;
\xc0\x00\x00\x00\x00\x00\x00\x00\xeaA0n\xeb4\x82-\xeaA0n\xebAU\xb1')
```

Coding the Water Sensor

Let's start out with the code that goes onto the FunHouse.

MQTT Secrets Settings

Since the code publishes directly to the MQTT server, there are a few more **secrets.py** file settings that the code expects to find. If your MQTT server has no username and password, you can change the value to `None`, however in general, the Home Assistant MQTT broker is set up to be password protected by default.

```
MQTT_BROKER = "192.168.1.1"
MQTT_PORT = 1883
MQTT_USERNAME = "myusername"
MQTT_PASSWORD = "mypassword"
```

To add code and libraries to your FunHouse, click the **Download Project Bundle** button to get the code and all of the libraries.

```
# SPDX-FileCopyrightText: 2017 Scott Shawcroft, written for Adafruit Industries
# SPDX-FileCopyrightText: Copyright (c) 2021 Melissa LeBlanc-Williams for Adafruit
Industries
#
# SPDX-License-Identifier: MIT

import os
import time
import board
import digitalio
import analogio
from displayio import CIRCUITPYTHON_TERMINAL
from adafruit_display_shapes.circle import Circle
from adafruit_funhouse import FunHouse

BOWL_STATE_TOPIC = "funhouse/catbowl/state"
LOW_VALUE = 4000
EMPTY_VALUE = 2000
UPDATE_INTERVAL = 1800  # Every 30 minutes

# Text labels for the Display
states = {
    "empty": "Add Water",
    "low": "Low",
    "full": "Full",
}

def publish_bowl_state(state):
    funhouse.peripherals.led = True
    # Publish the Bowl Level State
    print("Publishing to {}".format(BOWL_STATE_TOPIC))
    funhouse.network.mqtt_publish(BOWL_STATE_TOPIC, state)
    funhouse.peripherals.led = False

def connected(_client, _userdata, _result, _payload):
    status.fill = 0x00FF00
    status.outline = 0x008800

def disconnected(_client):
    status.fill = 0xFF0000
    status.outline = 0x880000

def get_bowl_reading():
    water_enable.value = True
    level = water_level_sensor.value
    water_enable.value = False
    return level

def get_bowl_state(level):
    if level <= EMPTY_VALUE:
        return "empty"
    elif level <= LOW_VALUE:
        return "low"
    return "full"

def bowl_level_display(level):
    if funhouse.peripherals.button_sel:
        return level
    return states[get_bowl_state(level)]

# Set Initial States
funhouse = FunHouse(default_bg=0x0F0F00)
funhouse.peripherals.dotstars.fill(0)
water_enable = digitalio.DigitalInOut(board.A0)
```

```
water_enable.switch_to_output()
water_level_sensor = analogio.AnalogIn(board.A1)
funhouse.display.root_group = CIRCUITPYTHON_TERMINAL
funhouse.add_text(
    text="Bowl Level:",
    text_position=(120, 60),
    text_anchor_point=(0.5, 0.5),
    text_color=0xFF0000,
    text_font="fonts/Arial-Bold-24.pcf",
)
level_label = funhouse.add_text(
    text_position=(120, 100),
    text_anchor_point=(0.5, 0.5),
    text_color=0xFFFF00,
    text_font="fonts/Arial-Bold-24.pcf",
)
funhouse.display.root_group = funhouse.splash

status = Circle(229, 10, 10, fill=0xFF0000, outline=0x880000)
funhouse.splash.append(status)

# Initialize a new MQTT Client object
funhouse.network.init_mqtt(
    os.getenv("MQTT_BROKER"),
    os.getenv("MQTT_PORT"),
    os.getenv("MQTT_USERNAME"),
    os.getenv("MQTT_PASSWORD"),
)
funhouse.network.on_mqtt_connect = connected
funhouse.network.on_mqtt_disconnect = disconnected

print("Attempting to connect to {}".format(os.getenv("MQTT_BROKER")))
funhouse.network.mqtt_connect()

last_reading_timestamp = None
last_bowl_state = None

while True:
    if (
        last_reading_timestamp is None
        or time.monotonic() > last_reading_timestamp + UPDATE_INTERVAL
    ):
        # Take Reading
        water_level = get_bowl_reading()
        # Update Display
        funhouse.set_text(bowl_level_display(water_level), level_label)
        # If changed, publish new result
        bowl_state = get_bowl_state(water_level)
        if bowl_state != last_bowl_state:
            publish_bowl_state(bowl_state)
            last_bowl_state = bowl_state
        last_reading_timestamp = time.monotonic()
```

Copy these over to the **CIRCUITPY** drive for your FunHouse board in the root directory along with your **secrets.py** file. The files on your board should look like this:

Code Walkthrough

Now to cover the code in sections. First are library imports. This includes the `FunHouse` library, the `Circle` to display if the board is connected, `time` for checking in intervals, and finally `board`, `analogio`, and `digitalio` for enabling and reading the sensor itself.

```
import os
import time
import board
import digitalio
import analogio
from adafruit_display_shapes.circle import Circle
from adafruit_funhouse import FunHouse
```

Next up is the MQTT topic, `BOWL_STATE_TOPIC`. The script will publish to the **state topic** if the reading has changed enough to let Home Assistant know the new state of the bowl.

`LOW_VALUE` is the raw analog reading for the sensor when the water is low. Anything above this value is considered full. `EMPTY_VALUE` is the raw analog reading for the sensor when the water is empty. Anything at or below this value is considered empty.

`UPDATE_INTERVAL` is the amount of time to wait in seconds for the FunHouse to check readings. By default this is once every half hour, which should be sufficient for something that changes very little over the course of a day.

```
BOWL_STATE_TOPIC = "funhouse/catbowl/state"
LOW_VALUE = 4000
EMPTY_VALUE = 2000
UPDATE_INTERVAL = 1800    # Every 30 minutes
```

This section contains a dict of different states. This is a convenient way to change the text that is shown on the display for each of these states.

```
# Text labels for the Display
states = {
    "empty": "Add Water",
    "low": "Low",
    "full": "Full",
}
```

This function will publish the current state of the bowl to MQTT, and thus Home Assistant. The output in this case is a raw text string value corresponding to one of the keys in the above dict that is published to the `BOWL_STATE_TOPIC`.

```
def publish_bowl_state(state):
    funhouse.peripherals.led = True
    # Publish the Bowl Level State
    print("Publishing to {}".format(BOWL_STATE_TOPIC))
    funhouse.network.mqtt_publish(BOWL_STATE_TOPIC, state)
    funhouse.peripherals.led = False
```

The next couple of functions are used for changing the circle to red or green depending on the connection status.

```
def connected(_client, _userdata, _result, _payload):
    status.fill = 0x00FF00
    status.outline = 0x008800

def disconnected(_client):
    status.fill = 0xFF0000
    status.outline = 0x880000
```

The code in the `get_bowl_reading()` function is meant to briefly enable the sensor, take a reading, and then shut it off. This is done to reduce electrolysis that may occur with the metal being in the water.

The code in the `get_bowl_state()` function just looks at the `level` value given to it, compares it to the thresholds that were set above and returns the state of the bowl.

The code in the `bowl_level_display()` function checks if the select button is held down and either returns a raw value or the text label for the current state.

```
def get_bowl_reading():
    water_enable.value = True
    level = water_level_sensor.value
    water_enable.value = False
    return level

def get_bowl_state(level):
    if level &lt;= EMPTY_VALUE:
        return "empty"
    elif level &lt;= LOW_VALUE:
        return "low"
    return "full"
```

```
def bowl_level_display(level):
    if funhouse.peripherals.button_sel:
        return level
    return states[get_bowl_state(level)]
```

The next bit of code creates a few of the variables with their initial states, including the `funhouse` object, the `water_enable` Digital IO, the `water_level_sensor` Analog IO and creates and draws the text labels.

```
# Set Initial States
funhouse = FunHouse(default_bg=0x0F0F00)
funhouse.peripherals.dotstars.fill(0)
water_enable = digitalio.DigitalInOut(board.A0)
water_enable.switch_to_output()
water_level_sensor = analogio.AnalogIn(board.A1)
funhouse.display.root_group = CIRCUITPYTHON_TERMINAL
funhouse.add_text(
    text="Bowl Level:",
    text_position=(120, 60),
    text_anchor_point=(0.5, 0.5),
    text_color=0xFF0000,
    text_font="fonts/Arial-Bold-24.pcf",
)
level_label = funhouse.add_text(
    text_position=(120, 100),
    text_anchor_point=(0.5, 0.5),
    text_color=0xFFFF00,
    text_font="fonts/Arial-Bold-24.pcf",
)
funhouse.display.root_group = funhouse.splash
```

This section initializes MQTT using the secrets, sets up the handler functions that were defined earlier, and connects. The `last_reading_timestamp` and `last_bowl_state` are then initialized and set to `None`. The `os.getenv()` function is used to get settings from settings.toml.

```
# Initialize a new MQTT Client object
funhouse.network.init_mqtt(
    os.getenv("MQTT_BROKER"),
    os.getenv("MQTT_PORT"),
    os.getenv("MQTT_USERNAME"),
    os.getenv("MQTT_PASSWORD"),
)
funhouse.network.on_mqtt_connect = connected
funhouse.network.on_mqtt_disconnect = disconnected

print("Attempting to connect to {}".format(os.getenv("MQTT_BROKER")))
funhouse.network.mqtt_connect()

last_reading_timestamp = None
last_bowl_state = None
```

Finally, there is the main loop. In the loop, it keeps looping and waits until the `UPDATE_INTERVAL` has passed. Once it has, it gets a reading and stores it in a variable so the sensor only has to be read once. It sets the display to show the appropriate message depending on the reading.

It next checks the state of the bowl and if the value differs from the last reading, it publishes the result to MQTT.

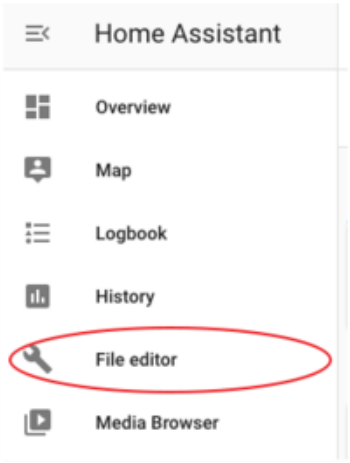
```
while True:
    if (
        last_reading_timestamp is None
        or time.monotonic() &gt; last_reading_timestamp + UPDATE_INTERVAL
    ):
        # Take Reading
        water_level = get_bowl_reading()
        # Update Display
        funhouse.set_text(bowl_level_display(water_level), level_label)
        # If changed, publish new result
        bowl_state = get_bowl_state(water_level)
        if bowl_state != last_bowl_state:
            publish_bowl_state(bowl_state)
            last_bowl_state = bowl_state
        last_reading_timestamp = time.monotonic()
```

Home Assistant Configuration

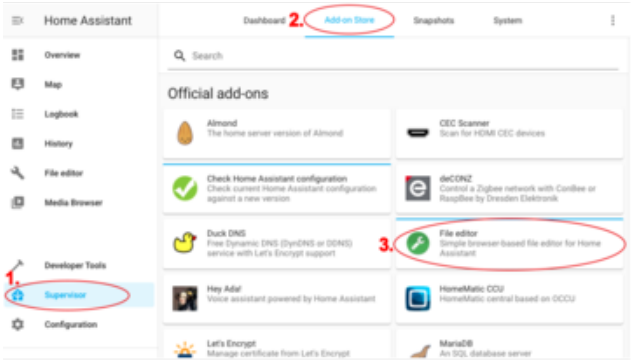
This guide assumes you already have a working and running Home Assistant server. If you don't, be sure to visit our Set up Home Assistant with a Raspberry Pi (https://adafru.it/lbd) guide first.

Check Your Add-Ons

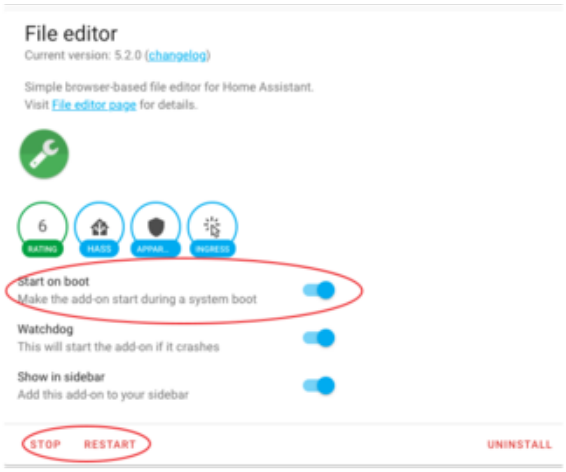
Start out by logging in and opening up your Home Assistant dashboard and checking that the File editor is installed.



As part of the setup, you should have an add-on either called **configurator** or **File editor** with a wrench icon next to it. Go ahead and select it.



If you don't see it, it may not be installed. You can find it under **Settings → Add-ons → Add-on Store → File editor** and go through the installation procedure.

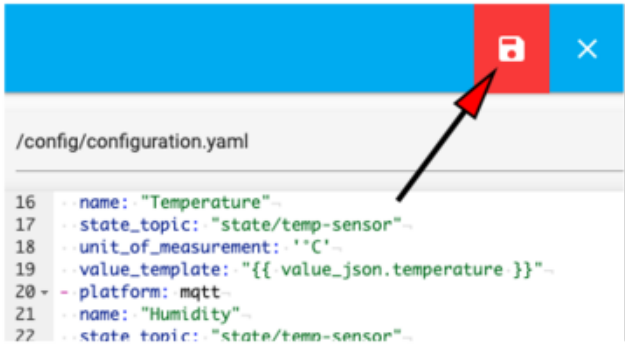


If you already have it, but it's just not showing up, be sure it is started and the option to show in the sidebar is selected.

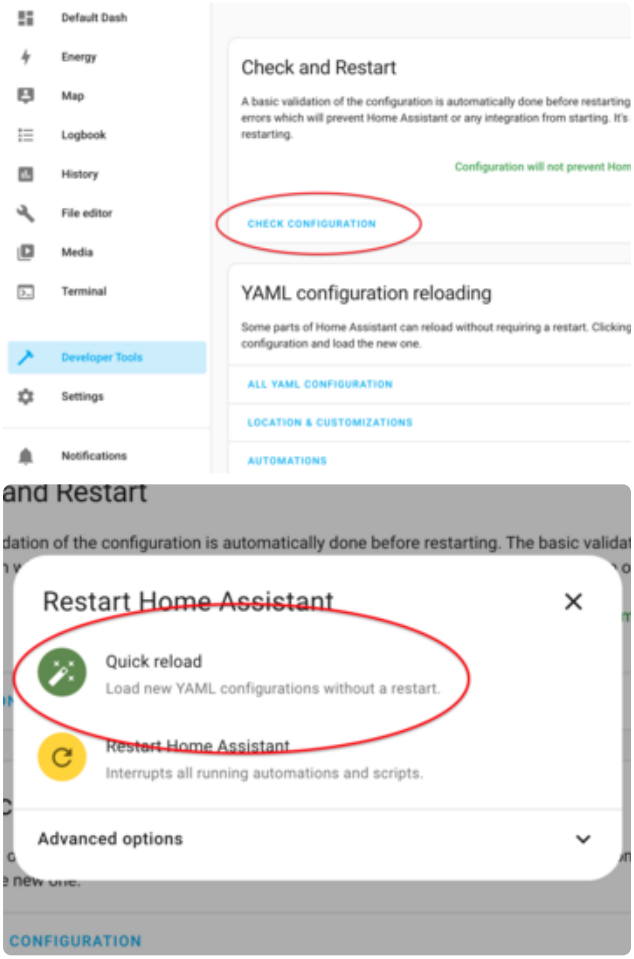
Adding the Sensor

In order to add the Water Level Sensor to Home Assistant, you'll want to add the following code to your configuration. This will create an MQTT sensor that listens on the `funhouse/catbowl/state` topic. It will translate the value it recives to one of the state labels. Feel free to update the states to your liking.

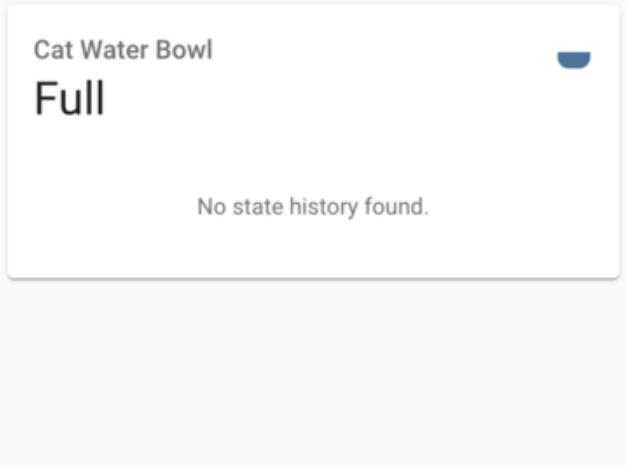
```
mqtt:
  sensor:
  - name: "Cat Water Bowl"
    unique_id: "funhouse_catwater"
    state_topic: "funhouse/catbowl/state"
    icon: mdi:bowl
    value_template:  &gt;-
      {% set states =  {
          'empty' : 'Add Water',
          'low' : 'Water Low',
          'full' : 'Full'
      } %}
      {{ states[value] if value in states else 'Unknown' }}
```



Click the save button at the top.



From the **Developer Tools** menu, you can check that the configuration is valid and click on **Restart** to load the configuration changes you made. You can just click **Quick reload** to reload any changes you made.



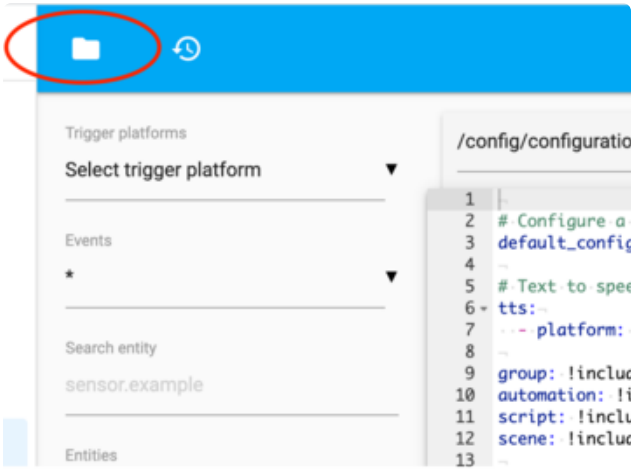
With the latest releases of Home Assistant, a LoveLace dashboard was added. If you haven't edited the Dashboard, it should automatically appear.

Otherwise, you may need to manually add a **Sensor** card to the dashboard.

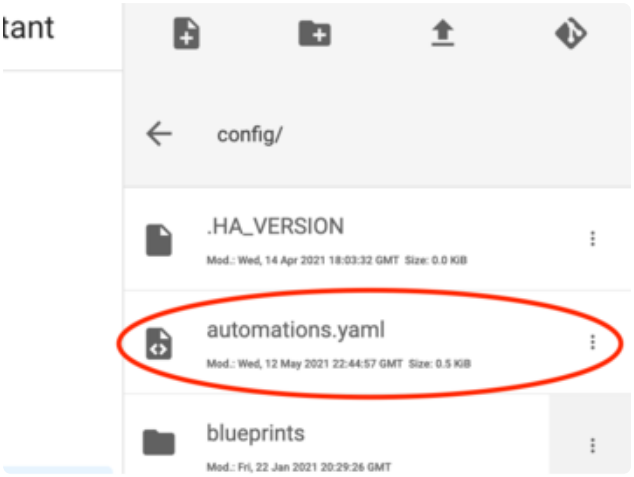
The **Cat Water Bowl** sensor should appear under your sensors.

Adding Notifications

This project wouldn't be complete without having Home Assistant notify you of when the bowl gets low or empty. Home Assistant uses Persistent Notifications as its main way of notifying you, which is what we'll be showing you how to add.



Start by going to the File Editor again and click the folder icon at the top.



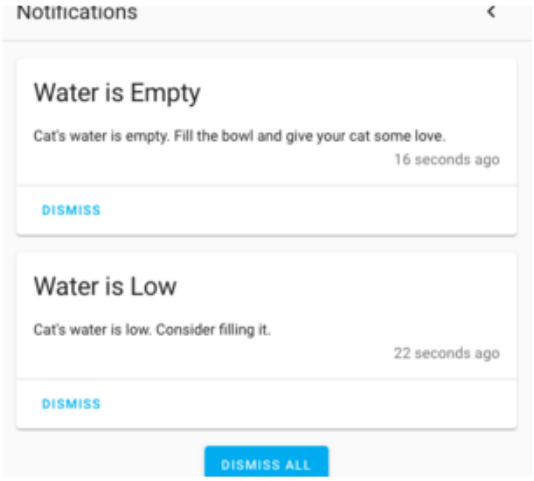
Open the **automations.yaml** file.

Add the following code. This will create 2 notification triggers. The first is when the state goes from **full** to **low**. The second is when it goes from any state to **empty**. Feel free to customize the messages.

```
- id: water_low_notif
  alias: "Cat Water Low"
  trigger:
  - entity_id: sensor.cat_water_bowl
    from: "full"
    platform: state
    to: "low"
  action:
    service: persistent_notification.create
    data:
      title: "Water is Low"
      message: "Cat's water is low. Consider filling it."

- id: water_empty_notif
  alias: "Cat Water Empty"
  trigger:
  - entity_id: sensor.cat_water_bowl
    platform: state
    to: "empty"
  action:
    service: persistent_notification.create
    data:
      title: "Water is Empty"
      message: "Cat's water is empty. Fill the bowl and give your cat some love."
```

Once that's added, go ahead and check the configuration and restart the server as you did before. If you made changes to the water bowl code above, you may need to update this code to reflect those changes.



To test alerts, you may need to temporarily increase the `UPDATE_INTERVAL` in your sensor code so it checks more frequently.

Troubleshooting

If you see the icons, but there is no data, it is easiest to start by checking the MQTT messages. Adafruit has a guide on how to use Desktop MQTT Client for Adafruit.io (https://adafru.it/kID), which can be used for the Home Assistant MQTT server as well.

Go ahead and configure a username and password to match your MQTT server and connect. Under **subscribe**, you can subscribe to the `#` topic to get all messages.

If you are seeing messages from the sensor, you may want to double check your Home Assistant configuration.

If you don't see any messages, you will want to follow the debugging section on the **Coding the Water Sensor** page.